A FORMAL DESCRIPTION OF BEHAVIORAL VERILOG BASED ON AXIOMATIC SEMANTICS

by

JOHN HOWARD ELI FISKIO-LASSETER

A THESIS

Presented to the Department of Computer
and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

August 1998

"A Formal Description of Behavioral Verilog Based on Axiomatic Semantics," a thesis prepared by John Howard Eli Fiskio-Lasseter in partial fulfillment of the requirements for the Master of Science degree in the Department of Computer and Information Science. This thesis has been approved and accepted by:

Dr. Amr Sabry

Date

Accepted by:

Dean of the Graduate School

An Abstract of the Thesis of

John Howard Eli Fiskio-Lasseter for the degree of Master of Science

in the Department of Computer and Information Science

to be taken August 1998

Title: A FORMAL DESCRIPTION OF BEHAVIORAL VERILOG BASED ON AXIOMATIC SEMANTICS

Approved: ______________________________

Dr. Amr Sabry

Reasoning about hardware designs written in Verilog is problematic, in large part because of the lack of a formal semantics for the language. The behavioral aspects of many constructs within the language are unclear, even with the existence now of an official language standard. As a result, a program may contain many subtleties which can be overlooked without careful analysis, and which may not even appear, depending on the implementation of the simulator that is used. In this thesis, we present a formal description of a large subset of behavioral Verilog, based on axiomatic semantics. Our primary contribution is an explicit formalization of the Verilog simulation cycle. In addition, we discuss some of the constructs that pose particular challenges to formal description, and offer axiomatic descriptions of these constructs that appear to match the behavior of the leading simulation packages.

CURRICULUM VITA

NAME OF AUTHOR: John Howard Eli Fiskio-Lasseter

PLACE OF BIRTH: Raleigh, North Carolina

DATE OF BIRTH: 29 December, 1968

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon
Earlham College

DEGREES AWARDED:

Master of Science, 1998, University of Oregon
Bachelor of Arts, 1992, Earlham College

ACKNOWLEDGEMENTS

My thanks first of all to my advisor, Dr. Amr Sabry. Without his exceptional inspiration and guidance, as well as the occasional kick in the pants (much needed), this project would not exist.

Thanks to Steven Sharp and Steve Meyer for their helpful correspondences and explanation of Verilog semantics. And thanks to the members of the Internet Verilog community of `comp.lang.verilog` who participated in my nonblocking assignment experiment—Hitesh Brahmbhatt, Larice Robert, Magnus Soderberg, Edward Arthur, and Robert Szczygiel.

Thank you to Daryl Stewart for reading this work, and for finding two small (but important) bugs.

Finally, a huge thank you to Janet, my life partner and closest friend—for being there.

DEDICATION

To my parents, who taught me to look, Bob Horn, who taught me how, and to Janet, who helped me to know what I saw.

TABLE OF CONTENTS

Chapter — Page

I. INTRODUCTION . . . 1

Related Work . . . 5

II. A BRIEF TOUR OF THE VERILOG LANGUAGE . . . 7

III. METHODOLOGY . . . 16

Finding a Suitable Grammar . . . 17
Normalization . . . 19
The Initial Transformation . . . 21

IV. NOTATION . . . 29

Simulation Cycle Phases . . . 30

V. DESCRIBING THE SEMANTICS OF VERILOG . . . 34

"Normal" Statements . . . 34
Reasoning About Side Effects *via* Blocking Assignment . . . 35
Non-Blocking Assignment . . . 42
Delay Controls . . . 48
Intra-Assignment Timing Controls . . . 54
Event Controls . . . 58

VI. CONCLUSIONS AND FUTURE WORK . . . 65

APPENDIX

A. SYNTAX . . . 69

Syntax of Behavioral Verilog Subset . . . 70
Evaluation Contexts . . . 71

B. AXIOMATIC DESCRIPTION OF VERILOG . . . 72

Ordinary Statements . . . 72
Access to the Store (Excluding Event Controls) . . . 72

Chapter Page

Non-Blocking Assignment 73
Scheduling of Delayed Threads 74
Event Controls 75

BIBLIOGRAPHY 76

CHAPTER I

INTRODUCTION

The Verilog hardware description language has become one of the leading tools for circuit designers. Like other simulators and simulation languages intended to aid in circuit design, Verilog supports a rich set of primitive constructs representing low-level digital design elements such as logic gates, transistors, and capacitive networks. What sets it apart from an ordinary circuit simulator, however, is the support for rapid prototyping using *behavioral* specifications of the same circuit. The hardware designer can use Verilog to specify a circuit using low-level structural elements, but she can also create a circuit with the same behavior by drawing from a large number of *high-level* programming constructs in syntax resembling C, such as loops, assignment, conditional statements, and so on. Further, Verilog is a *parallel* programming language, and thus provides the usual complement of tools for the parallel programmer (guards, semaphores, *etc.*).

First and foremost, however, Verilog is a *simulation-driven* language, which is to say that the semantics of its constructs are largely defined by a standardized discrete event execution model. In order to really understand the semantics of

Verilog constructs, therefore, one must understand them with respect to this model. Thus a thorough understanding of the behavior of the simulator is crucial. As the *IEEE Standard* states ([12], p.45): "Although the Verilog HDL is used for more than simulation, the semantics of the language are defined for simulation, and everything else is abstracted from this base definition."

In this thesis, we propose a formal description of behavioral Verilog that also includes an explicit formalization of this simulation cycle. This formalization is implemented as an *axiomatic* semantics, an approach which we believe offers a number of advantages. First of all, the axiomatic approach is more *abstract*. Since we reason only by transformation of Verilog programs into other Verilog programs, our semantics do not resemble any particular implementation—higher order logic, duration calculus, etc.—too closely. This offers the possiblity of greater *flexibility* and thus a more *general-purpose* description.

Further, the axiomatic approach corresponds to a common way in which most Verilog users think about the language. The typical life cycle for a Verilog design is roughly the following: a prototype of the circuit is rapidly created using Verilog's high-level behavioral constructs, and after some testing to verify that the overall black-box approach is sound, the circuit is re-written at a lower-level of abstraction, and tested again. Finally, when all of the circuit has been written in a sufficiently structural description (usually at the logic-gate level), a netlist can be synthesized from the code, the results tested against the original program for behavioral equivalence, and (if successful) given to a fabrication tool which produces a corresponding IC die.

Often, these transformations are performed manually, rendering from a behavioral specification of a circuit an equivalent structural description. In this way, the Verilog engineer comes to know the semantics of the high-level constructs in terms of more primitive ones. The point here is that this is essentially the way an axiomatic semantics works. Consequently, the axiomatic approach is in many ways the most *intuitive* approach to formal description: there is no need to understand any complex mathematical objects, as with an operational description, nor any denotational models.

In addition to providing a foundation for the application of various formal methods to Verilog programs, the specification of a formal semantics for Verilog can bring to light many subtle points of the language that may be overlooked otherwise. Verilog began as (and remains today) an industrial language whose design was geared toward providing a practical tool for engineers rather than a tool for academic research. Until the release of the *IEEE Standard* [12] in 1996, there was no official standard semantics for the language. At the time of this writing, no complete *formal* semantics exists at all. Moreover, there is still, to our knowledge, no existing formal description of the Verilog language that explicitly captures the simulation cycle itself or describes the behavioral constructs in terms of this cycle. As a result, there are many constructs within Verilog whose behavior can vary subtly from simulator to simulator; one cannot simply look at a Verilog program and know what it does without running it.

This point must be emphasized: it is not merely a statement about the presence of race conditions, which are a hazard of concurrent languages in general, nor is it simply a point about the theoretical mismatch between the expressive power of behavioral versus structural constructs in Verilog[1]. What is meant here is that without a formal semantics to precisely define every aspect of a construct's behavior, different Verilog simulators can give subtly different results on the same piece of code. This can occur even if both simulators correctly implement the same semantics: without the precision of a formal description, however, such differences may not be easily detectable.

Some of these subtleties are discussed in this thesis, along with the development of each construct's formal representation. The goal of our work is to provide a general-purpose formal representation of Verilog's behavioral constructs and of the behavior of the Verilog simulation cycle in a manner that expresses as accurately as possible each construct's "standard" semantic definition, yet also corresponds in a plain fashion to the ordinary human intuitions about the language. In the following chapter, we describe the basic methodology.

---

[1]There are behavioral programs in Verilog for which no corresponding structural version—and hence no circuit—can exist. This is known as the *model mismatch problem*, and is discussed by Szu-Tsung Cheng [4] in the introduction of his doctoral dissertation.

Related Work

Research by the members of the Verilog Formal Equivalence Project (VFE) at Cambridge has had a strong influence on the present work. Many of the subtleties of the language represented in our semantics are described in Gordon's "Semantic Challenge" paper [9], and it is from this work that this author first learned about Verilog. Gordon's recent work has been directed toward the development of different theoretical models for understanding Verilog and other HDL semantics ([8], [10]), using subsets of the language with simplified semantics. Stewart is investigating the semantic properties of many structural constructs, particularly port connections and continuous assignment ([16], [15]).

Another prominent effort in formal reasoning about behavioral Verilog is the work surrounding the VIS system and its FSM compiler, `vl2mv` ([5], [6], [7]). VIS has successfully synthesized and verified substantial Verilog programs, and is even capable of performing some optimizations on the resulting logic. The formal semantics developed for the language are only implicitly given by the transformation algorithms, however. Formal descriptions of a program's timing properties and concurrent behavior are actually performed on the resulting FSMs generated by the compiler.

Pace [13] has recently published a paper detailing a formal semantics for Verilog, but with slightly simplified behavior. In this paper, he gives an operational semantics, based on a variant of the discrete duration calculus, as part of a larger effort to develop a unified framework for simulation and formal verification.

VHDL is a hardware description language similar in character to Verilog. Van Tassel [17] gives an operational semantic description of its simulation cycle, formulated in the higher-order logic notation of the Cambridge HOL system.

The only other axiomatic semantic description of a hardware description language of which we are aware is a short paper by Hua and Zhang [11], in which they describe an axiomatic semantics for the *Iowa Logic Specification Language*.

CHAPTER II

A BRIEF TOUR OF THE VERILOG LANGUAGE

A circuit can be specified in Verilog using low-level constructs that represent common circuit elements such as logic gates, transistors, and intra-circuit wiring. In addition, one can specify the desired timing properties of each component. The resulting design can then be tested using the Verilog event-driven simulator or even formally verified via a number of commercial and public-domain products. For example, a simple one-bit adder (with carry-in and carry-out), might be designed at the gate-level like this:

```
module halfadder(x,y,sum,car);
   input x,y;
   output sum,car;

   assign #1 sum =  x ^ y;
   assign #1 car = x & y;
endmodule
```

```
module fulladder(x,y,carryin,sum,carryout);
   input x,y,carryin;
   output sum,carryout;
   wire s1,c1,c2;

   halfadder h0(x,y,s1,c1);
   halfadder h1(s1,carryin,sum,c2);
   assign #1 carryout = c1 | c2;
endmodule
```

All Verilog programs, are composed of one or more *modules*, in one sense the highest-level construct in a program. The manner in which one module is connected to another is specified by the designer. From the point of view of the containing `module`, each contained `module` can be thought of as a black box, once we have established its input/output and timing behavior. In addition to the instantiation of sub-modules (if any), a `module` consists of some number of *threads*, each of which executes concurrently with all the other threads of all instantiated modules in a program. Threads may run continously, such as those declared with `always` or `assign`, or be declared `initial`, and run only once.

The data type of variables in Verilog programs can be roughly classified in two ways: *register* and *net*. Register variables include traditional types such as `integer` or `real`, or they can be declared `reg`, which simply corresponds to a variable whose size (in number of bits) can be explicitly declared.

Net types include `wire` variables, such as those declared in our adder, as well as a number of other types. Representing the physical connections within a circuit, they do not hold any data, but simply “carry” a value from one point in the circuit to another. In other words, we can inspect the current value of a net variable, but traditional procedural assignment cannot be applied to it. The only allowable “assignment” is the representation of a wiring connection, either at declaration or with the use of `assign`, which is known as *continuous assignment*. The basic idea behind a continuous assignment is that it is specified exactly once, and then, whenever any part of the right hand side of the expression changes value, the entire right hand side is reevaluated and the result becomes the new value of the left hand side. Occasionally, as in our example, one may specify a constant-valued delay, which causes the update to the net variable to be delayed for the specified number of clock cycles, each time the right hand side changes.

Let us now create another version of the same halfadder:

```
module halfadder_beh(x,y,sum,car);
   input x,y;
   output sum,car;
   reg sum,car;

   always @(x or y)  begin
      if (x == 1) begin
        if (y == 0) begin
           sum <= #1 1;
           car <= #1 0;
        end else begin
           sum <= #1 0;
           car <= #1 1;
        end
      end else begin
         car <= #1 0;
         if (y == 1)  sum <= #1 1;
         else         sum <= #1 0;
      end
   end
endmodule
```

This second version demonstrates a number of behavioral constructs, including the use of an *event control*. A *guard* ("`@(x or y)`") is set to watch for any change in x or y, and it blocks further execution of the thread at this point. These changes are also called *edges*, and event controls are therefore referred to as *edge-sensitive*. Whenever there is an edge, the logical AND and XOR of `x` and `y` are re-computed.

We write out directly the procedure for these computations, using conditional statements and a sequence of procedural assignments (`<=`). Actually, Verilog supports two different types of procedural assignment. Traditional procedural assignment may be specified using the `=` operator, and is known in Verilog as *blocking* *assignment.* Blocking assignment corresponds to the way we normally think of procedural assignment: when one is encountered, the value of the right hand side is computed and the register variable is immediately updated with this value, which it holds until the next update.

By using the `<=` operator instead, we use *nonblocking assignment*, which behaves like blocking assignment with one important difference: when one is encountered, the value of the right hand side is computed, but the update to the register variable is not performed until the end of the current clock cycle. Until that time, the variable retains its value from the beginning of the clock cycle (unless changed by a blocking assignment, of course). If, for instance, `a` is 1 and `b` 0 at the beginning of a simulation cycle, then

```
a <= b;
b <= a;
```

will cause `a` to be 0 and `b` 1 at the end of that cycle.

In order to represent the timing behavior of a component, Verilog provides a procedural *delay control* using the `#` operator. The `#` operator may be used with any expression as an argument: the effect, when one is encountered, is to evaluate this expression and schedule a delay of the thread for the number of clock cycles specified by the expression's value. When used at the beginning of a statement, the effect of a delay control is to block further execution of a thread at the beginning of this statement, until the simulation clock has advanced to the point specified by the delay.

Both delay and event controls may also be used in a timed procedural assignment, known as *intra-assignment delay*, although it is more generically termed *intra-assignment timing control*. The effect of such a timing control is radically different, depending on whether it is used with a blocking or nonblocking assignment.

With an intra-assignment delay on a blocking assignment, which is of the form `x = #`e_1 e_2, the effect is to evaluate e_1 and e_2 immediately, and schedule at the time specified by e_1 an update to `x` of the value computed for e_2. Further execution of the thread is blocked until this point. The only real difference between this kind of intra-assignment delay and a delayed blocking assignment statement (`#`e_1 `x =` e_2) is that with the ordinary delay, e_1 is evaluated, and then the thread is blocked: e_2 will not be evaluated until the thread is reawakened at a later time. The use of an *event control* has similar properties, the only differences arising from the standard distinctions between delay and event control release of a thread.

If we use a nonblocking assignment instead (`x <= #`e_1 e_2), e_1 and e_2 are immediately evaluated as before, and a nonblocking update is scheduled to be performed on `x` at the time specified by the delay. The crucial difference here is that further execution of the thread is not blocked, but continues right away, with `x` retaining its old value. The use of an event control merely puts off the execution of the nonblocking assignment until the specified guard can fire, and also allows continued execution of the original thread.

In all cases, Verilog also supports the concept of a δ-time delay (*i.e.*, an event that will happen in the current clock cycle, but guaranteed later than the other events currently active), by using a `#0`.

These timing controls provide some ordering constraints on the threads in a program: all uncontrolled threads happen before zero-delayed threads, and both happen before any threads that are blocked by a guard or scheduled (with a delay) to another time slot.

In fact, a single cycle of a Verilog simulation consists of five phases, which execute in the following order([12], section 5):

1. The *active events*. These are "ordinary threads"—the threads that are not blocked by either delay or event controls of any kind. If there are any such threads scheduled to run in this cycle, they are run, in arbitrary order.

2. The *delayed events*. All threads that were blocked with a zero delay are made active at this point, and run in arbitrary order. Note that it is possible to block a thread with more than one `#0`, and that threads blocked by multiple zero-delays will still execute in the current cycle, but *after* all threads blocked with fewer zero-delays.

3. *Non-blocking updates.* In this phase, all updates scheduled by non-blocking assignments are executed, in the order in which the original assignments occurred. Note that if the nonblocking assignment was made with an intra-assignment delay, the update is not performed here, but rather during the *non-blocking updates* phase of a later clock cycle.

4. The *monitor events.* These are the events corresponding to the Verilog constructs that begin with the `$` symbol, such as `$display`. When such a construct is encountered in a thread, the corresponding monitor event is scheduled to run at this phase of the cycle, after all active, inactive, and non-blocking update events have run. Monitor events execute in the order they were scheduled. These constructs are not really part of a program execution, but rather are used to display information about the state of computation at a certain point. We will not, therefore, consider them further in our formalization of the simulation cycle.

5. *Future events.* These are the threads that are either blocked by a guard or delayed to run at some future time. When there are no more active or delayed threads, and no more non-blocking updates, the simulation clock is advanced until one of the events can be run. The *IEEE Standard* divides future events into two classes—*future inactive events* and *future nonblocking assignment update events*—which correspond, respectively, to those events which will occur

during the *active events* phase of a later cycle and those which will occur during the *nonblocking updates* phase.

Although not directly stated in this section of the *Standard*, we can also classify the *future inactive events* as being either *guarded* or *delay-blocked*. Our axioms use this distinction explicitly.

CHAPTER III

METHODOLOGY

The idea elaborated in this thesis is to develop a formal description consisting of a set of *reduction axioms*. For each syntactic construct there are associated with it one or more axioms, each of which specifies a legal reduction of that construct to another one. Although this thesis does not directly use the results of either work, we acknowledge previous work by Boehm [2] and a paper by Sabry and Field [14] as foundational and inspirational sources.

Each axiom describes formally the behavior of a construct S_1 in terms of a (usually) simpler construct, S_2, by stating that any occurrence in a program of S_1 can be rewritten as S_2[1].

For example, we can describe the behavior of *if-then* statements as follows:

$$\begin{array}{lcll} \texttt{if (} n \texttt{) } S & \longrightarrow & S & \bullet \text{if } n \neq 0 \\ \texttt{if (0) } S & \longrightarrow & \epsilon & \end{array}$$

[1]The intuition here is similar to the sort of reductions that are carried out in elementary algebra: "$\sqrt{x^2} \longrightarrow x$", for example.

This says that an *if-then* statement that executes S only on condition of *true* (*i.e.*, a non-zero value) can be rewritten as S itself, and that the execution of S on condition of the value *false* can be deleted altogether[2]. With this approach we are able to define a "symbolic execution" of Verilog programs, through the repeated evaluation of program elements. Notice how both of these axioms require that the $\langle exp \rangle$ in the conditional evaluation be a constant value. Notice further that since these are the only two axioms whose left-hand side can be matched to *any* form of *if-then* statement, the conditional expression itself must first be evaluated completely. Otherwise, no further reduction is defined.

Finding a Suitable Grammar

The Verilog language is enormously rich and feature-laden, and to attempt a complete account of every syntactic element would introduce a complexity that would quickly overwhelm other efforts. In papers concerning formal semantics or formal reasoning about Verilog, therefore, it is typical to treat only a *subset* of the full language specification, and the present work is no exception.

The aim of this work is to provide a general-purpose formal description of the behavioral subset of the language, and of the simulation cycle. Hence our main criteria for a language subset is that it be a large enough subset of real Verilog to

---

[2]Throughout our notation, we make frequent use of the null character, ϵ, typically in order to indicate that something has been deleted. Note that ϵ is *not* a syntactic element of the language: it is merely a notational device to indicate a null or empty element.

be representative of the most useful constructs in the language, yet small enough not to obstruct the work of describing the simulator.

We will choose, therefore, a "core" subset of the full syntax consisting only of some representative behavioral elements. In particular, we explicitly *exclude* system constructs (*i.e.*, those causing *monitor events*) as well as the non-standard additions to the language provided by many vendors.

Further, we exclude the "structural elements" of the language, such as continuous assignment (with or without delay), capacitive nets, and gate or transistor-level primitives. This may seem to be a strange limitation, but again, our goal is to provide a formalization of the behavioral constructs of Verilog, along with the behavior of these constructs within the simulation cycle. For reasons that are discussed briefly in the conclusion, the formalization of the structural components of the language is a different problem from that of formalizing the behavioral components. Introducing them would likely require additional techniques that are unnecessary in accounting for behavioral modeling. This is an area that requires further work before it can be successfully integrated into the present semantic description.

The complete grammar of our behavioral subset is given as an Appendix.

Normalization

In order to make the task of reasoning about a Verilog program tractable, we will first perform a number of normalizing transformations to each module.

The first normalization is a *flattening transformation*, borrowed from Gordon [9], by which we collect into a single top-level module the set of threads to execute concurrently. The basic procedure consists of (i) renaming the variables local to each module instance to avoid clashes (ii) replacing module instances by the instantiated threads that they contain, and finally (iii) redeclaring all local variables at top level. This last part includes the variables that are local to labeled blocks. At this point, we can also delete the labels themselves, since we are not including the `disable` construct in our subset.

Gordon observes (p.8) that after flattening, the only way that a wire can be driven is by a continuous assignment. Since we do not, in the present work, include continuous assignment at all, we can eliminate wires entirely. Since wires only exist to connect ports together, a bottom level module can have only a `reg` variable, r, as a source on an `output` port. Thus, the `wire`, w, connecting to this port from the containing module is guaranteed to be connected to a register. Ordinarily, this port connection would be replaced by a continuous assignment (*i.e.*, `assign w = r;`). Here, we can simply replace each occurrence of w with r. This leaves us with a `reg` variable that can only be connected across modules as either (i) the source on an enclosed module's `input` port or (ii) the source on the enclosing module's `output` port.

When this process has been repeated on all enclosed modules, the result is a new bottom level module (the original enclosing one, now flattened) that only has `reg` variables as the sources on its output ports. The flattening can now be continued at the next higher level. At the very top level, if the module contains no ports, we will have no wires left. If it does contain ports, those declared as `input` will be wires, to which we can simply assign initial constant values.

Note that this elimination will not work if a `wire` is connected to more than one output port sink, since this results in a logic conflict. For present purposes, we will disallow such connections.

The elimination will also fail if we allow continuous assignment, since at this point we risk the possibility of feedback. This is the case where the expression on the right hand side of a continuous assignment refers, either directly or indirectly, to the net variable on the left hand side. For example:

```
assign x = z & y;
assign y = x;
```

In such cases, the attempt to replace occurrences of the left hand side by the right will result in an infinite loop. On the other hand, the elimination procedure is also undesirable if we wish to include continuous assignment.

The Initial Transformation

Simple constructs such as `if (`e`)` S are easy to represent in an axiomatic fashion because all the information we care about (*i.e.*, whether S executes) is directly representable within Verilog. Other behavioral aspects of Verilog—including the simulation cycle—pose a number of challenges, however.

For representing the contents of memory at any given point, we can use a sequence of ordinary blocking assignments to constant values, one for each declared register variable. So that it can easily be identified as the block representing the contents of the store, we define a special labeled environment for it (which we know is unique, since we have removed all other labels during normalization). Thus, for a module with `reg` variables `a`, `b`, and `c`, whose current values are 1, 0 and `x` (the *unknown* value), we would include in our module the following thread:

```
initial
   begin: _ST
      a = 1;
      b = 0;
      c = x;
   end
```

In fact, we will include not one but *two* representations of the store, ST_2 and ST_1, which represent (respectively) the most recent update to each variable and the contents of the store just before the current set of updates. The reason for this concerns the problem of representing *edges*, and is discussed in detail in Chapter V.

Of course, the computational state of a Verilog program does not just include the contents of the store. In addition to this, we must also provide an explicit formal representation of all the threads on the event queue at any given point.

Here, the real challenge is to represent the threads of execution in each of the various phases of the cycle in such a way that we guarantee an order of execution corresponding to that specified in the standard ([12], section 5). How we do it depends on how we choose to represent each phase of the cycle.

To begin, we will represent the pending *nonblocking updates* in a similar fashion to that of the store: as a labeled sequential block of non-blocking assignments to constant values. Suppose, for example, that we perform two non-blocking assignments to a variable `a` which evaluate to 0 and 1, followed by an assignment to `b` of 1 and another assignment to `a` of 0. Then we would represent the scheduled nonblocking updates as:

```
initial
   begin: _NBU
      a <= 0;
      a <= 1;
      b <= 1;
      a <= 0;
   end
```

For the moment, let us defer represention of the threads in the various other phases of the cycle. Given our choice of representation for the store and the nonblocking updates, the ordering constraints of the four main cycle phases, which we detailed above, require that we guarantee the following ordering of execution:

1. The "old" copy of the store—*i.e.*, ST_1—which represents the state of memory with which the current program begins execution.

2. The scheduling of *non-blocking updates*. Literally speaking, these are the non-blocking assignments in the `_NBU` block.

3. Those threads that belong to the *future threads* phase, both delay-blocked and guarded. These are all the threads that were blocked at the point of execution represented by the program.

4. The "current" copy of the store—*i.e.*, ST_2—which represents the state of memory after all the updates that were made just at the point of execution represented by the current program.

5. The *active threads*.

6. The *delayed threads*.

Some explanation for this is obviously in order. The execution of ST_1 before anything else is obvious, since this is the initial state. Actually, we can also consider the execution of the nonblocking updates block to happen first: it does not much matter which of these is first or second, since nonblocking updates will not be carried out until later on in the cycle anyway. What we are doing here is making sure that all the updates which were already scheduled at the start of this computation will in fact be scheduled before any other updates which should come later.

Perhaps the strangest claim is that the future threads must be run *immediately after* (1) and (2), and *before* any of the current updates to the store, the active threads, and the delayed threads. Consider, however, what a *future thread* really is: it is a thread that begins with either a delay or event control. In other words, when we run these threads, each one will block, as it should. The importance of this occurring lies really with the *guarded* threads. Remember that the future threads are those threads that have already been blocked. In the case of the event controls, this means that if the execution of the second copy of the store (ST_2) causes a change to any variable on which a guard is waiting, the guard should be able to see it and wake its thread appropriately. If the guards have not already been evaluated (and placed in the *future threads* phase), these changes will be missed.

Given this argument, the remainder of the ordering constraints should be clear. (1) through (3) set up the complete computational state, as it stands at the start of a program's execution. (4) then executes all the changes to the store that "were about to happen" at the start of the computation. (5) and (6) then correspond to the rest of program that has not yet been executed at the start of this computation.

There may be a number of ways to guarantee this ordering, but we will make use here of the δ-delay (`#0`) for its intuitive clarity:

1. Neither the ST_1 nor the *nonblocking updates* blocks shall have any delay on them.

2. Likewise, those *future threads* that are blocked by *delay controls* shall remain unmodified (remember that these will block immediately anyway, so the use of a `#0` is pointless).

3. The *future threads* blocked by *event controls* will have a single `#0` just before their event control. This ensures that no guard accidentally fires when the updates in ST_1 are performed.

4. ST_2 will have two consecutive δ's (`#0#0`) inserted before the first assignment.

5. All other threads will have three δ's (`#0#0#0`) inserted before each one's first statement. Notice that this means that the *delayed threads* will in fact begin with at least *four*, not three δ's.

Now we may complete our normalization process from the previous chapter. Suppose we now have a flattened, label-and-wire-free single module:

```
module M1;
  reg a,b;

  always  statement₁
  initial statement₂
endmodule
```

First, normalize each `always` and each `initial` construct by first transforming each `always` thread into an equivalent `initial`. Then enclose each thread body in a sequential block that begins with a 3-δ delay:

```
module M1;
  reg a,b;

  initial begin #0#0#0 forever statement₁ end
  initial begin #0#0#0 statement₂ end
endmodule
```

The lack of distinction between `initial` and `always` threads may seem a bit strange, but it is justified by the fact that `always` S is semantically equivalent to `initial forever` S[3].

Now add the *nonblocking updates* block (NBU) which, of course, is empty for now (we haven't started the program yet):

```
module M1;
  reg a,b;

  initial begin: _NBU end
  initial begin #0#0#0 forever statement₁ end
  initial begin #0#0#0 statement₂ end
endmodule
```

Lastly, add the representation of the store, by inserting the ST_1 and ST_2 blocks into the module and adding to both of them exactly one assignment of the *unknown*

[3]Steven Sharp, personal communication. See also Gordon [9].

value, `x` to each declared variable. Since each block only assigns to a variable once, the order of the assignments does not matter. The final, transformed program, is:

```
module M1;
  reg a,b;

  initial begin: _ST1 a = x; b = x; end
  initial begin: _ST2 #0#0 a = x; b = x; end
  initial begin: _NBU end
  initial begin #0#0#0 forever statement_1 end
  initial begin #0#0#0 statement_2 end
endmodule
```

Notice that we initialize variables to the Verilog *unknown value*, `x`, in accordance with the official specification ([12], §3.2.2), but that there are other choices. We could just as easily choose another set of initialization values, for instance, if we wanted to examine the behavior of the port variables in a module under different valuations.

Under the transformations specified, we assert that in all cases, the transformed programs exhibit behavior that is *identical* to the originals and hence they are equivalent. The reason for this is fivefold. First, the initial transformation never adds any threads in a *future threads* representation, since before the program begins execution, there are no such threads. Second, the initial addition of the *nonblocking updates* thread has no effect on a program, since it is always added as an empty sequential block. Third, the ST_1 and ST_2 blocks are added in such a way that they perform identical assignments that correspond to the initialization values specified by the *Standard*, and therefore neither one effects any net change to the memory state before the program proper (*i.e.*, the *active threads*). By these three arguments, therefore, we can see that no change to the computational state can occur before the original program threads can begin execution.

Now for the *active threads* themselves. Notice that the addition of a `begin-end` to each of the original program threads has no effect: this is the fourth point. Finally, although the addition of three δ's to the beginning of each thread obviously effects a slight change in the simulator's scheduling behavior, it affects every thread in the same way. Clearly, therefore, the ordering properties of the original threads are preserved.

Nonetheless, a literal axiomatic description of the resulting programs threatens to be quite confusing, requiring very long axioms in order to capture the syntactic elements which indicate the properties of each construct about which we wish to reason. In order to make this task manageable, we introduce some notation, to which we now turn.

CHAPTER IV

NOTATION

What we have created so far is an explicit representation of the full computational state of a behavioral Verilog program within Verilog itself. For the sake of some clarity, we now introduce several pieces of notation, in which the axiomatic description itself will be written.

Some of the notation is fairly straightforward. We use ϵ, for example, to represent the empty construct. It is not a syntactic element, of course, but merely a notational reminder, typically indicating that something has been deleted. The reduction symbol $\longrightarrow$ is used to indicate that a reduction may be made between the left and right hand sides; that is, the left hand side may be re-written into the form specified on the right hand side. Some of our axioms group elements together with { }, in order to enhance readability.

In order to shorten our axioms a bit, we will also use quite a few notational devices whose appearance within the axioms resembles real syntactic elements. THREAD [ ], is the generic representation of a thread body, and a THREADLIST indicates zero or more threads within a particular cycle. We will also find it useful to use the shorthands UPDATES and NBUPDATES, which are simply sequences of blocking (*resp.* nonblocking) assignments to variables of constant values. Other "pseudo-syntax" notation is a bit more complicated, and is explained in the following sub-sections.

Simulation Cycle Phases

The various cycle phases introduce by far the greatest complexity into our notation. Since most of our axioms will need to account for the complete computational state, we specify a shorthand for this: ST, AT, DT, NBU, FD, and FG, which correspond (respectively) to the *store*, the *active threads*, the *delayed threads*, the *nonblocking updates*, and the two classes of *future thread*: *delay-blocked* and *guarded*. Notice that we ignore the classification of *future nonblocking assignment update events*, which exists in the *Standard* ([12], §5).

The first and second copies of the store are denoted by ST_1 and ST_2.

In all cases, however, remember that this is only *notation*, not syntax. At bottom, we are still working only via transformation from Verilog programs to Verilog programs. The order in which we write each phase shorthand does not imply that these threads occur in the source text in any order at all (such order is irrelevant in a parallel language): it is simply a concise way of representing the relevant information about the source text contents. If a particular phase is relevant to some axiom, it is included there. If its contents are not relevant, the shorthand is typically omitted from the axiom. Each of these pieces of notation corresponds to the set of threads in the source text that match a particular syntactic form:

- ST_1 and ST_2 are both sequential blocks, one labeled `_ST1`, and the other `_ST2`. `NBU` likewise is a sequential block with the label `_NBU`.

- `FD` are, literally, the threads that begin with a constant-valued, non-zero delay. `FG` are the threads that begin with exactly one `#0`, followed immediately by an event control.

- `AT` are those threads that begin with exactly three occurrences of `#0`, and `DT` are those threads with more than three.

In some of the side conditions of our description, we make use of the ϵ symbol to indicate the absence of any threads in a particular phase, as in "$\mathsf{AT} = \epsilon$" (read: there are no *active threads*).

The occurrence of $\mathsf{ST}_1 = \mathsf{ST}_2$ in a side condition denotes a requirement that all of the assignments to same variables be consistent between each copy of the store (which means that no edges are currently present).

Evaluation Contexts

We make use of the notation "$\Gamma[\sigma]$" to denote an occurrence of σ within context Γ. Two of these contexts—$\mathcal{E}$ and $\mathcal{G}$—specify evaluation order within expressions and guards, and are part of the axiomatic description itself. They are formally defined in Appendix A.

We will also use this notation *informally* to indicate the occurrence of an element within one of the cycle phases. For example, "$[\sigma]$" on either ST_1 or ST_2 should read: "the occurrence of the assignment σ in ST_1 (or ST_2).

Further, the use of $[\sigma]$ with a given phase, such as $\mathsf{AT}[\sigma]$ denotes the occurrence of σ as the body of a THREAD, which is itself executing in parallel with the other threads in that phase. See also the following...

Parallel Execution

"$\alpha \parallel \beta$" denotes the execution of α and β in parallel. Occasionally, we will also use this notation with a *phase* Φ (for example, $\mathsf{AT} \parallel \alpha$) to denote the execution of a THREAD (or even an entire THREADLIST) α in parallel with all the other threads in phase Φ. An important point to stress about "$\alpha \parallel \beta$" is that no guarantee is made regarding the order of execution or the interleaving of threads.

Representation of Thread Scheduling

Obviously, the same thread will be represented in different ways, depending on whether it is *active*, *zero-delayed*, *delay-blocked*, or *guarded*. In most cases, however, we are not interested in the particulars of this representation, and need only to state that a thread has been rescheduled somewhere else. The particulars of the representation are only interesting in developing the appropriate data structures for each phase.

If we wish to indicate the change of a thread's status from *guarded* to *zero-delayed*, for example, the actual re-write would involve replacing the event control on the beginning of the thread with four δ's:

$$\texttt{initial begin \#0 @(}\langle \mathit{guard}\rangle\texttt{)}\ \langle \mathit{stat}\rangle\ \texttt{end}$$
$$\longrightarrow\quad \texttt{initial begin \#0\#0\#0\#0}\ \langle \mathit{stat}\rangle\ \texttt{end}$$

This is cumbersome and unnecessary when we need to give a complete formal definition, however, so we use instead:

$$\mathsf{FG} \parallel \mathsf{THREAD}\ [\ \texttt{@(}\langle \mathit{guard}\rangle\texttt{)}\ S\ ] \quad\longrightarrow\quad \mathsf{AT} \parallel \mathsf{THREAD}\ [\ \texttt{\#0}\ S\ ]$$

which conveys the necessary information in much less space.

CHAPTER V

DESCRIBING THE SEMANTICS OF VERILOG

“Normal” Statements

The constructs in our subset of Verilog can be roughly classified according to whether they reference or alter the store and according to whether their execution includes any timing controls. Those that do neither one are “normal” procedural statements, such as conditional execution, discussed above. In general, these constructs are the easiest to describe, as their semantics are well-understood. They are:

(1)	$\texttt{if (} n \texttt{) } S$	$\longrightarrow$	S	$\bullet$ if $n \neq 0$
(2)	$\texttt{if (0) } S$	$\longrightarrow$	ϵ	
(3)	$\texttt{if (} n \texttt{) } S_1 \texttt{ else } S_2$	$\longrightarrow$	S_1	$\bullet$ if $n \neq 0$
(4)	$\texttt{if (0) } S_1 \texttt{ else } S_2$	$\longrightarrow$	S_2	
(5)	$\texttt{while (}\langle exp \rangle\texttt{) } S$			
	$\longrightarrow$ $\texttt{if (} \langle exp \rangle \texttt{) begin } S\texttt{; while (}\langle exp \rangle\texttt{) } S \texttt{ end}$			
(6)	$\texttt{forever } S$	$\longrightarrow$	$\texttt{while (1) } S$	
(7)	$\epsilon;\ S$	$\longrightarrow$	S	
(8)	$\mathsf{THREAD}\ [\ \texttt{begin } S\texttt{; } \langle statlist \rangle \texttt{ end}\ ]$			
		$\longrightarrow$	$\mathsf{THREAD}\ [\ S\texttt{; begin } \langle statlist \rangle \texttt{ end}\ ]$	
(9)	$\mathsf{THREAD}\ [\ \texttt{begin } \epsilon \texttt{ end } S\ ]$			
		$\longrightarrow$	$\mathsf{THREAD}\ [\ S\ ]$	
(10)	$\mathsf{THREAD}\ [\ \epsilon\ ]$	$\longrightarrow$	ϵ	

<u>Reasoning About Side Effects *via* Blocking Assignment</u>

Because we exclude the `function` construct from our subset, expressions cannot produce changes to the store. As a result, the task of axiomatizing side effects and lookup is greatly simplified.

In our semantics, the primitives representing the memory state are the assignments of constant values to variables that occur in ST_1 and ST_2, which, because of the behavior defined by the axioms, are the most recent values assigned to each variable at any point in the computation.

The axioms concerning the store fall into three categories: update of a variable's value, lookup of a variable's value, and the presence or absence of an edge in each update (for event controls). Specifically, the following facts about side effects in Verilog must be expressed:

1. When an assignment is made to a variable x, the location in memory represented by x must be updated to the new value. However,...

2. If an update to x causes a certain kind of change from x's old value, then every event control guarding on this kind of change *must* be released before any further updates can be made to x.

3. When a variable x is encountered in the evaluation of an expression, its current value in memory may be substituted for any occurrence of x in the expression, but the value of x in memory is never changed as a result of this.

In our description, the simplest task lies in describing updates. Since changes to the store can only happen within statements, we need only concern ourselves with axiomatizing *assignment*. Verilog supports two kinds of assignment—blocking and nonblocking—and each may be executed with or without an intra-assignment timing control. In our semantics, however, the only type of assignment that can effect changes to the store is blocking assignment without any timing control—specifically, those blocking assignments that occur in the *active threads* phase of the simulation cycle (AT). This is sufficient because, as we shall see, the other three types of assignment are really just blocking assignments whose timing controls enforce a certain order to their execution.

At a first pass, then, we might axiomatize updates as:

$$\mathsf{ST}[x \texttt{ = } n_1]\ \mathsf{AT}[x \texttt{ = } n_2\texttt{;}\ S] \quad\longrightarrow\quad \mathsf{ST}[x \texttt{ = } n_2]\ \mathsf{AT}[S]$$

This is not sufficient, however, to accomodate the constraint given above in (2), since it offers us no way to detect edge changes. Our approach to this is the double representation of the store using ST_1 and ST_2. In this scheme, ST_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 ST_1.

With this approach, we have an explicit representation not only of the store, but also of any rising or falling edges on each variable. It introduces yet another subtlety, however, which is best explained by giving the final form of the single axiom governing side effects in our semantics:

$$(20)\quad \mathsf{ST}_1[x = n_1]\ \mathsf{ST}_2[x = n_1]\ \mathsf{AT}[x = n_2\,;\ S] \quad\longrightarrow\quad \mathsf{ST}_1[x = n_1]\ \mathsf{ST}_2[x = n_2]\ \mathsf{AT}[S]$$

Notice that this axiom can only be applied to the case where the value of x is the same in both ST_1 and ST_2 (*i.e.*, no edges are present on x). With this requirement, we ensure not only that the proper value is written to x, but also that if an edge occurs, it cannot be deleted while there are still guards that could fire on it.

Of course, we then need to specify how such an edge is cleared and the update committed to both ST_1 and ST_2. This and other matters pertaining to event controls are discussed below.

Expression Evaluation

As with most procedural languages, the rules that define *how* expressions are evaluated in Verilog are extremely complex, but generally uninteresting. Two points are worth noting, however.

First, all operations on variables are modulo 2^n, where n is the bit-size of the variable. This is true of most languages, of course, but in Verilog, the bit-size of many variables can be explicitly declared, using an optional range specification[1].

[1]Without a range specification, variables of type `reg`, `wire`, or one of several others, are assumed

Variables can be accessed bit-by-bit, using the operator `[`i`]`, which accesses the i^{th} bit. This means that attention must be paid to the range declaration of a variable when determining the modulus of operations on it.

Secondly, each digit of a variable can take on one of four values: `1`, `0`, `z` (high-impedance), or `x` (unknown value). A description of the semantics of various operators requires a truth-table-like exhaustion of possible combinations over all four values. Because of this "four-valued" logic, the usual rules for arithmetical and boolean operations on variables increase substantially in complexity.

We would, for example, describe the semantics of the "`==`" operator thus:

n_1 `==` n_2	$\longrightarrow$	`1`	• if each bit in n_1 and n_2 the same, and neither number contains an `x` or `z`
n_1 `==` n_2	$\longrightarrow$	`0`	• if any bit in n_1 and n_2 differs, and neither number contains an `x` or `z`
n_1 `==` n_2	$\longrightarrow$	`x`	• if either number contains an `x` or `z`

...and so on. As is apparent from this, the semantics of expression evaluation involve a large number of very similar axioms whose specification is tedious and whose complexity is unproductive for present purposes. Hence, we will avoid a full description in this thesis, referring the reader instead to Section 4 of the *IEEE Standard* [12].

There are some general properties of expression evaluation that we will include here, however. We can specify *how* an expression is evaluated (*i.e.*, the order of

---

to be 1 bit in size. Variables of `integer`, `real`, etc. have the usual size conventions.

evaluation) by defining an *evaluation context* for expressions, $\mathcal{E}$. Other than operator precedence rules, the *Standard* does not specify any particular order: for the sake of simplicity, we will assume here that it is left-to-right. This is not an axiom, of course, but belongs instead with the syntax of the language:

$$\begin{array}{rcl} \mathcal{E} & ::= & [\,] \\ & | & \langle u\_op \rangle\ \mathcal{E} \\ & | & \mathcal{E}\ \langle b\_op \rangle\ \langle exp \rangle \\ & | & \langle \mathsf{NUMBER} \rangle\ \langle b\_op \rangle\ \mathcal{E} \\ & | & \mathcal{E}\texttt{?}\ \langle exp \rangle \texttt{:}\ \langle exp \rangle \end{array}$$

As suggested in the previous section by constraint (3), most expression evaluations share in common a need to reference the value of one or more locations in memory. A reasonably complete semantics should at least include rules that define this. For the sake of readability we define nine, each of which corresponds to expression evaluation in a different context:

(11) $\mathsf{ST}_2[x \texttt{ = } n]\ \mathsf{AT}[\texttt{if (} \mathcal{E}[x] \texttt{) } S]$
$\longrightarrow\ \mathsf{ST}_2[x \texttt{ = } n]\mathsf{AT}[\texttt{if (} \mathcal{E}[n] \texttt{) } S]$

(12) $\mathsf{ST}_2[x \texttt{ = } n]\ \mathsf{AT}[\texttt{if (} \mathcal{E}[x] \texttt{) } S_1 \texttt{ else } S_2]$
$\longrightarrow\ \mathsf{ST}_2[x \texttt{ = } n]\ \mathsf{AT}[\texttt{if (} \mathcal{E}[n] \texttt{) } S_1 \texttt{ else } S_2]$

(13) $\mathsf{ST}_2[x_1 \texttt{ = } n]\ \mathsf{AT}[x_2 \texttt{ = } \mathcal{E}[x_1]]$
$\longrightarrow\ \mathsf{ST}_2[x_1 \texttt{ = } n]\ \mathsf{AT}[x_2 \texttt{ = } \mathcal{E}[n]]$

(14) $\mathsf{ST}_2[x_1 \texttt{ = } n]\ \mathsf{AT}[x_2 \texttt{ = } \langle delay\_or\_event\_control \rangle\ \mathcal{E}[x_1]]$
$\longrightarrow\ \mathsf{ST}_2[x_1 \texttt{ = } n]\ \mathsf{AT}[x_2 \texttt{ = } \langle delay\_or\_event\_control \rangle\ \mathcal{E}[n]]$

(15) $\mathsf{ST}_2[x_1 \texttt{ = } n]\ \mathsf{AT}[x_2 \texttt{ = \#}\mathcal{E}[x_1]\ \langle exp \rangle]$
$\longrightarrow\ \mathsf{ST}_2[x_1 \texttt{ = } n]\ \mathsf{AT}[x_2 \texttt{ = \#}\mathcal{E}[n]\ \langle exp \rangle]$

$$(16)\quad \mathsf{ST}_2[x_1 \texttt{ = } n]\ \mathsf{AT}[x_2 \texttt{ <= } \mathcal{E}[x_1]] \\ \longrightarrow\ \mathsf{ST}_2[x_1 \texttt{ = } n]\ \mathsf{AT}[x_2 \texttt{ <= } \mathcal{E}[n]]$$

$$(17)\quad \mathsf{ST}_2[x_1 \texttt{ = } n]\ \mathsf{AT}[x_2 \texttt{ <= } \langle \mathit{delay\_or\_event\_control} \rangle\ \mathcal{E}[x_1]] \\ \longrightarrow\ \mathsf{ST}_2[x_1 \texttt{ = } n]\ \mathsf{AT}[x_2 \texttt{ <= } \langle \mathit{delay\_or\_event\_control} \rangle\ \mathcal{E}[n]]$$

$$(18)\quad \mathsf{ST}_2[x_1 \texttt{ = } n]\ \mathsf{AT}[x_2 \texttt{ <= \#}\mathcal{E}[x_1]\ \langle \mathit{exp} \rangle] \\ \longrightarrow\ \mathsf{ST}_2[x_1 \texttt{ = } n]\ \mathsf{AT}[x_2 \texttt{ <= \#}\mathcal{E}[n]\ \langle \mathit{exp} \rangle]$$

$$(19)\quad \mathsf{ST}_2[x \texttt{ = } n]\ \mathsf{AT}[\texttt{\#}\mathcal{E}[x]\ S] \\ \longrightarrow\ \mathsf{ST}_2[x \texttt{ = } n]\ \mathsf{AT}[\texttt{\#}\mathcal{E}[n]\ S]$$

$$(20)\quad \mathsf{ST}_1[x \texttt{ = } n_1]\ \mathsf{ST}_2[x \texttt{ = } n_1]\ \mathsf{AT}[x \texttt{ = } n_2\texttt{;}\ S] \ \longrightarrow\ \mathsf{ST}_1[x \texttt{ = } n_1]\ \mathsf{ST}_2[x \texttt{ = } n_2]\ \mathsf{AT}[S]$$

In effect, they all state that an expression context containing a variable x may replace that occurrence of x by its value in the second (current) copy of the store, without altering that value.

Note that unlike the single axiom governing *writes*, we relax the requirement that the value of x be consistent between ST_1 and ST_2. In a language like C, in which expressions can themselves cause side effects, this would result in an incorrect description of the semantics of event controls, for the reasons discussed above: we might cancel a change in a variable's value before all relevant guards can fire on it. Our subset of Verilog does not permit this behavior, however.

Note further that *the* value of an expression in a certain state is not necessarily well-defined. Consider the following (somewhat silly, but legal) code:

```
initial begin x = 1;  if ((!x) || x) some_statement; end
initial x = 0;
```

Will `some_statement` execute? In almost every case, yes, but we cannot guarantee it. Suppose we first evaluate the assignment of `x` to 1, applying Axiom 20, followed immediately by an application of Axiom 11 to the `!x` portion of the *if*-expression (which will evaluate to 0). Now since there are no guards to fire (see V.6, below), we apply the axiom that allows us to clear the rising edge from memory (Axiom 32), and then apply Axiom 20 to the assignment of `x` to 0. Finally, we apply Axiom 11 once more in order to evaluate `x`; only now, `x` has the value 0 in memory, so the whole expression evaluates to `(0 || 0)`, and `some_statement` will not execute.

It is doubtful that one could find a Verilog simulator that did *not* guarantee execution of `some_statement`. Although we were unable to verify this firsthand, the Verilog-XL package from Cadence reportedly implements the notion of *time token movement*, which prevents the interleaving of statements that execute as a single block between timing controls[2]. Since almost every Verilog simulator follows the behavior defined by Verilog-XL, one could reasonably expect never to see this kind of interleaving in practice.

Time-token movement would certainly be sufficient to prevent another thread from changing the value of a variable in the middle of an expression evaluation, and since it would be more in the spirit of real Verilog simulators, we could choose to reflect this in our semantics. However, it is still not clear that we could prevent

[2]Steve Meyer, personal communication.

intra-assignment, once we introduce the `fork/join` construct, which executes in one thread and runs the statements within its body concurrently. Further, the introduction of the `function` construct would introduce the possibility that expression evaluation could itself produce side effects, which again leaves us vulnerable to this interleaving.

Regardless of the practicalities of implementation, the *Standard* itself is fairly clear on the matter:

> Another source of nondeterminism is that statements without time-control constructs in behavioral blocks do not have to be executed as one event... At any time while evaluating a behavioral statement, the simulator may suspend execution and place the partially completed event as a pending active event on the event queue. The effect of this is to allow the interleaving of process execution. Note that the order of interleaved execution is nondeterministic and not under the control of the user ([12], §5.4.1)

Thus, we choose to implement a conservative, "worst case" semantics, which allows full interleaving of statements and expression evaluation. In truly simultaneous execution of the two threads above, one cannot *guarantee*, in a formal sense, that `some_statement` executes, and this is what our semantics reflects.

Non-Blocking Assignment

Non-blocking assignment (NBA) is supported in Verilog by using the `<=` operator instead of `=`. It is often used in Verilog programs to model the way register transfers work in some systems: in general, its effect is to assign to a variable at the end of a time unit the value of the right-hand side of the assignment from the beginning of the time unit.

Other than the time at which the updates actually occur, however, non-blocking assignment has exactly the same effect on the store as ordinary blocking assignment, and our axioms defining its behavior reflect this:

(21) $\mathsf{AT}[x$ `<=` n ; $S]$ {`initial begin: _NBU` $\mathsf{NBUPDATES}_i$ `end`}
  $\longrightarrow$ $\mathsf{AT}[S]$ {`initial begin: _NBU` $\mathsf{NBUPDATES}_i$ x `<=` n;`end`}

(22) ST_1 ST_2 AT DT {`initial begin: _NBU` $\mathsf{NBUPDATES}_i$ `end`}
  $\longrightarrow$ ST_1 ST_2 {THREAD [$\mathsf{UPDATES}_i$] $\|$ AT} DT
  `initial begin: _NBU` ϵ `end`

- if $\mathsf{AT} = \mathsf{DT} = \epsilon$, and $\mathsf{ST}_1 = \mathsf{ST}_2$
- each nonblocking assignment in $\mathsf{NBUPDATES}_i$ is transformed to a blocking assignment in $\mathsf{UPDATES}_i$, according to the schema :
  x `<=` n ; $\longrightarrow$ x `=` n ;

In english, (21) says that a NBA of a constant value to `x` schedules an update to `x` that will occur immediately after all the other non-blocking updates already scheduled. Notice how we require that the right-hand side be a constant value, which ensures that it is fully evaluated before the update is scheduled. Axiom 22 then says that when all active and delayed threads have been run, and when there are no more rising edges that could trigger a guard, the sequence of nonblocking updates occurs. Actually, the requirement that $\mathsf{ST}_1 = \mathsf{ST}_2$ is unnecessary, and it is included only for intuitive clarity. Since the updates execute as blocking assignments in an *active thread*, Axiom 20 already ensures that any edges that should be detected will be.

Notice, by the way, that unlike most of the other representational transformations involved in moving a thread between different phases, the transformation of the body of the NBU block into an active thread is non-trivial. Axiom 22 is really an axiom *schema*, which a specifies a simple transformation to be applied to each assignment in the NBUPDATES block. This is one of three axiom schemas in our semantics (the other two are Axioms 24 and 25).

This brings up, however, two important subtleties of ordinary NBA that are easy to overlook. The first point, which is more of an explanation than a new observation, is that when a nonblocking update to a variable is scheduled, it *must* be performed, even if another nonblocking update to that same variable is scheduled immediately afterwards. This is, in fact, stated quite clearly in section 5.4.1 of the *Standard*: "Nonblocking assignments shall be performed in the order the statements were executed."

Without a thorough understanding of the reasons for this, however, one might be tempted (as this author first was) to optimize the nonblocking updates phase by keeping only the last scheduled update to each variable. This will not do, however, because *nonblocking updates can themselves change the computational state.* Consider the following:

```
module nbupdate_test;
   reg [1:0] x, y;
   initial #10 $stop;
   always @(posedge x) y = x;
   initial y = 2;
   initial begin
      x = 0;
      #5 x <= 1; x <= 0;
   end
endmodule
```

A simulator that executes every scheduled nonblocking update will cause `y` to get the value 0 at the end of the 5^{th} time step. On the other hand, a simulator that (incorrectly) attempts to optimize away the earlier update will fail to trigger the waiting `posedge x` guard, and `y` will have the value 2 throughout all 10 clock cycles.

Within industry, this does not appear to be a common bug, however. The Veriwell free simulator (v2.1.7), to which we had direct access, sets y to 0. An informal collaborative experiment, in which this code was run on a number of different simulator packages by participants of the USENET group `comp.lang.verilog`, suggests that most all of the commercially-available packages also implement this behavior correctly, as shown in Table 1.

The exceptions are the FinSim and Chronologic simulators, although when `x` is changed to a 1-bit register, FinSim sets y correctly.

As an interesting aside, note that although the `posedge x` is triggered, the body of the statement never sees `x` with the value 1, since this value has been

TABLE 1. Results of `nbupdate_test`.

Simulator	Version	Reported Result
Veriwell	2.1.7	0
SuperFINSIM	4.5.11	2
Modelsim	5.1d	0
Verilog-XL	2.6.18	0
ChronologicVCS	4.4.1	2
NC-Verilog	1.22(s37)	0

overwritten by the later nonblocking update. In the Verilog-XL and NC-Verilog simulators (both from Cadence Systems), this is the occasion for an (optional) explicit change to the semantics [3]: the `+delay_trigger` command line option changes the semantics of event controls so that a guard is only triggered if the process waiting on the guard actually sees the condition being waited on. If this option is used, both simulators leave `y` with the value 2. Further, both simulators reportedly are optimized to only schedule the last nonblocking update in a sequence of assignments, if they come from the same assignment statement. For example, the `@(negedge r)` guard in the following code will *not* fire :

[3] Steven Sharp, personal communication.

```
always @(negedge r) statement
initial begin
  i = 0;
  while (i <= 1) begin
     r <= i;
     i = i+1;
  end
end
```

The second point about nonblocking assignment—a consequence of the first—is that the completion of the scheduled non-blocking updates does not necessarily mean the end of the current cycle. This is specified in the *IEEE Standard* ([12], §9.2.2), albeit somewhat confusingly:

> *At the end of the time step* means that the nonblocking assignments are the last assignments executed in a time step—with one exception. Nonblocking assignment events can create blocking assignment events. These blocking assignment events shall be processed after the scheduled nonblocking events.

What this appears to mean, based on the previous discussion, is this: because nonblocking updates can cause waiting guards to fire, it is possible for a series of nonblocking updates to make one or more threads active again, *in the same cycle.* It is therefore not only *blocking assignment events* but *any* event that can be made active in the same time step, but *after* the scheduled nonblocking updates.

It is for this reason that Axiom 22 specifies that the UPDATES in the NBU phase be executed as a series of blocking assignment in a single *active* thread: in effect, these updates create the possibility that the cycle can start over again, without advancing the simulation clock. The following code, for example, will cause an infinite loop at simulation time 5:

```
initial begin
   #5 x = 0;
 end

always @(negedge x) begin
    x <= 1; x <= 0;
 end

always @(posedge x) begin
    x <= 0; x <= 1;
 end
```

When `x` is assigned 0 at time 5, the `negedge x` guard fires, causing two non-blocking assignments to be scheduled. When these updates are carried out, they will in turn release the waiting posedge guard, which schedules two more non-blocking updates, causing the `negedge` guard to fire once again, and so on. This behavior is similar—though not identical—to that of an infinite loop described by Gordon ([9], pp.3-4).

Delay Controls

Of the two types of Verilog timing controls, delays are the simplest to describe, because their timing properties are more readily predictable. Delay controls are used in two ways: at the beginning of a statement, and in *intra-assignment* delays.

Delays used in the "ordinary" way always serve to halt continued execution of a thread until a later time step. Non-zero delays suspend a process until a later tic of the simulation clock. A δ or zero-value delay, on the other hand, still causes a process to execute in the current time step, but serves to guarantee some ordering of execution between processes.

Non-Zero Delays

When a statement of the form `#`n S is encountered in a thread, the effect is to suspend execution of the thread at this point. Then, when the current time step is finished, the simulation clock is advanced until the earliest delay scheduled in any thread is reached. All such threads that were delayed to this point are made *active* and run, in arbitrary order.

Equivalently, we can say that when `#`n S is encountered, it is simply suspended. Then, when the current time step has finished, the simulation clock advances *by one tic*, and we evaluate the thread again, only this time as `#(`n`-1)` S . This process of evaluation/suspension is repeated, until there is no delay left, at which point the thread is able to continue execution at S. This interpretation is expressed by two axioms:

(23) $\{\mathsf{THREAD}\ [\ \texttt{\#}n\ S\ ] \parallel \mathsf{AT}\}\ \ \mathsf{FD} \ \longrightarrow\ \mathsf{AT}\ \ \{\mathsf{FD} \parallel \mathsf{THREAD}\ [\ \texttt{\#}n\ S\ ]\}$
$\bullet$ if $n > 1$

(24) $\mathsf{ST}_1\ \mathsf{ST}_2\ \mathsf{AT}\ \mathsf{DT}\ \mathsf{NBU}\ \{\mathsf{FD} \parallel \mathsf{THREADLIST}_i\}$
$\longrightarrow\ \mathsf{ST}_1\ \mathsf{ST}_2\ \{\mathsf{THREADLIST}_i \parallel \mathsf{AT}\}\ \mathsf{DT}\ \mathsf{NBU}\ \mathsf{FD}$
- if $\mathsf{AT} = \mathsf{DT} = \mathsf{NBU} = \mathsf{FD} = \epsilon$, and $\mathsf{ST}_1 = \mathsf{ST}_2$
- the body of each THREAD in $\mathsf{THREADLIST}_i$ is transformed to an *active thread* representation, according to the schema :
  $\mathsf{THREAD}\ [\ \mathtt{\#}n\ S\ ]\ \longrightarrow\ \mathsf{THREAD}\ [\ \mathtt{\#(}n\mathtt{-1)}\ S\ ],\ \mathit{if}\ n > 1$
  $\mathsf{THREAD}\ [\ \mathtt{\#1}\ S\ ]\ \longrightarrow\ \mathsf{THREAD}\ [\ S\ ]\ ,\ \mathit{otherwise}$

Axiom 23 states that a thread whose body begins with a constant-value delay greater than 1 is added to the *future threads* phase of the cycle, unchanged. Literally speaking, the only change we make to the source text is to remove the three #0's that mark the thread as *active.*

The final axiom (Axiom 24) states that once there are no more *active*, *delayed*, or *non-blocking update* events to run (*i.e.*, once the current cycle is really over), the threads suspended by delays will be re-evaluated (in arbitrary order) as *active* threads. The effect of this on the computational state is the same as advancing the simulation clock to the next time at which a thread can become active, except that we don't need to keep track of the clock time. Note that the side condition $\mathsf{FD} = \epsilon$ ensures that *all* of the *delay-blocked* threads are made active at the advance of the clock: it is impossible to choose a THREADLIST that is only a proper subset of FD.

Axiom 24 is another axiom *schema*, and the specified representational transformation of each thread is easy, but important. Notice that we do not permit a statement of the form #1 S to be re-released as #0 S, because a thread scheduled to wake up in the current cycle becomes part of the *active threads* phase, not *delayed.*

Actually, this is a distinction without a difference, in some sense. The behavior of delayed threads would be just as correct if we used the schema:

$$\mathsf{THREAD}\ [\ \mathtt{\#}n\ S\ ] \ \longrightarrow\ \mathsf{THREAD}\ [\ \mathtt{\#(}n\mathtt{-1)}\ S\ ],\ if\ n > 0$$

so that the thread is finally released with a zero-delay.

The reason for this is that if we get to a point in the computation where we are able to apply (24), it means (by assumption of the side conditions for the axiom) that no other threads except for those in FD can run. Consequently, the threads released from FD are guaranteed to have the earliest execution time. If the least possible delay on any thread in this group is `#0` instead of no delay at all, then such threads will still be the first to execute in the cycle. We would therefore have a *de facto* implementation of the proper semantics, even though, strictly speaking, it would be incorrect.

On the other hand, our optimization, in addition to being more in the spirit of the true semantics, saves us from having to repeatedly apply the axioms concerning `#0` each time we "advance the clock".

As in Axiom 22, we also require that ST_1 and ST_2. In this case, however, the side condition is not merely illustrative but *necessary* for correct execution. Without it, we may inadvertently release a thread delayed to the next cycle while there are still guarded threads that can run in this one. Suppose for instance that we add **`#1 some_statement`** to our infinite loop example from the previous discussion of NBA:

```
initial begin
   #5 x = 0;
   #1 some_statement;
 end

always @(negedge x) begin
   x <= 1; x <= 0;
end

always @(posedge x) begin
   x <= 0; x <= 1;
end
```

Because time step 5 will never finish in this example, `some_statement`, which is delayed to time 6, will never execute. If we don't require that ST_1 and ST_2 be consistent, however, then we have no effective way to determine that no more guards can fire. Thus we could, incorrectly, apply Axiom 24 and allow `some_statement` to run.

Zero-Delays

Zero-delays (`#0`), also called *δ-delays*, corresponds to an event that happens at no quantifiable distance away on the clock, but nonetheless happens "later than now". It is typically used as a way of forcing determinacy of execution between threads, as in

```
always @(posedge clk) #0 x = y;
always @(posedge clk) y = y + 1;
```

which guarantees that the assigment to x is always made after `y` increments its previous value.

Were we to need a formal definition of the effect of a `#0` on a thread's scheduling behavior (*i.e.*, the change in a thread's status from *active* to *zero-delayed*), we would use an axiom along the following lines:

$$\{\mathsf{THREAD}\ [\ \texttt{\#0}\ S\ ]\ \|\ \mathsf{AT}\}\ \ \mathsf{DT}\ \longrightarrow\ \mathsf{AT}\ \ \{\mathsf{THREAD}\ [\ S\ ]\ \|\ \mathsf{DT}\}$$

In our representation of a thread's status, however, this would mean only that we deleted the `#0` from the beginning of the thread body, and then in changing from *active thread* to *zero-delayed* thread representation, we replaced the initial `#0#0#0` with `#0#0#0#0`. In other words, this axiom does nothing whatsoever under our semantics, because no *active thread* body can begin with a `#0`! Any active thread that begins with a `#0` actually begins with at least four of them. Consequently, it already *is* a *zero-delayed* thread, and we do not need to reason about it further at this point.

On the other hand, we do need to define the transition from *zero-delayed* to *active*. This is accomplished by the following axiom:

(25) $\mathsf{ST}_1\ \mathsf{ST}_2\ \mathsf{AT}\ \ \{\mathsf{DT} \,\|\, \mathsf{THREADLIST}_i\}$
$\quad\longrightarrow\ \mathsf{ST}_1\ \mathsf{ST}_2\ \ \{\mathsf{THREADLIST}_i \,\|\, \mathsf{AT}\}\ \ \mathsf{DT}$
- if $\mathsf{AT} = \mathsf{DT} = \epsilon$, and $\mathsf{ST}_1 = \mathsf{ST}_2$
- each THREAD in $\mathsf{THREADLIST}_\mathsf{i}$ is transformed to an *active thread* representation, according to the schema :

```
initial begin #0#0#0#0 S end
     ⟶ initial begin #0#0#0 S end
```

Axiom 25 is the third of our three axiom *schemas*, and is quite similar in appearance to Axiom 24, which governs the release of *nonzero-delay* blocks. However, the representational transformation of each thread is actually trivial, since it involves nothing more than the ordinary rewrite of the initial four δ's to three. It is included largely for the sake of clarity.

In a similar fashion to 24, we require here that the DT matched is in fact empty. In other words, if this axiom can be applied all, it must be applied to the entire THREADLIST that is currently classified as *zero-delayed.* Here, too, the requirement that all edges be cleared from the state ($\mathsf{ST}_1 = \mathsf{ST}_2$) is essential, for the same reasons given for the release of *nonzero delays.*

Intra-Assignment Timing Controls

Blocking Assignment with Timing Control

Of the two kinds of intra-assignment timing control, *blocking assignment* is the simplest, containing few surprises. As mentioned previously, the difference between a *blocking assignment with timing control* (x = $C\ e$) and a *timing controlled blocking* *assignment* ($C\ x$ = e) is in the time at which the right hand side of the assignment is evaluated. With an intra-assignment control, e is evaluated immediately, as is the event control C, and the thread is then blocked until the time specified by C. At this point, x will get the value that was computed for e, and the thread will continue execution. With a timing controlled assignment, on the other hand, C will be evaluated, and then all further evaluation of the statement is suspended until the specified time. At this point, the thread will be re-activated, e will be evaluated, and x will immediately get this value.

If e is itself a constant expression, then the two types of timing control are equivalent in every respect. Thus, we can define an intra-assignment timing control on blocking assignment quite easily:

$$(26)\quad x \texttt{ = } \langle \mathit{delay\_or\_event\_control} \rangle\ n\,\texttt{;} \quad \longrightarrow \quad \langle \mathit{delay\_or\_event\_control} \rangle\ x \texttt{ = } n\,\texttt{;}$$

This, of course, effectively requires that the right hand side be fully evaluated before the axiom can be applied, as it should.

Non-Blocking Assignment with Timing Control

The single axiom governing *nonblocking assignment* is not much more complicted than its cousin:

(27) $\mathsf{AT}[x$ `<=` $\langle \textit{delay\_or\_event\_control} \rangle\ n\,;\ S]$
$$\longrightarrow\ \{\mathsf{AT}[S] \,\|\, \mathsf{THREAD}\ [\ \langle \textit{delay\_or\_event\_control} \rangle\ x \texttt{ <= } n\,;\ ]\}$$

In words, this states that the occurrence of a nonblocking assignment with a timing control schedules a nonblocking update of n to be made to x at the point specified by the $\langle \textit{delay\_or\_event\_control} \rangle$, and that further execution of the original thread continues without interruption. The continued execution of the thread is the obvious difference between the use of nonblocking versus blocking assignment.

There is, however, a more obscure difference here that our axiom reflects. Multiple non-blocking assignments with timing controls can lead to unexpected race conditions, *even if the order of the original statements was well-defined.*

The account of this in the *Standard* is brief, but clear: "When multiple non-blocking assignments are scheduled to occur in the same register in a particular time slot, the order in which the assignments are evaluated is not guaranteed—the final value of the register is indeterminate" ([12], p.102).

The following code, for example, results in a race condition on `x`:

```
initial begin
   x <= #4 1;
   x <= #4 0;
end
```

and so does:

```
initial begin
   x <= @(posedge clk) 1;
   x <= @(posedge clk) 0;
end
```

Morevover, this is the reason that `x <= #0 y;` is *not* necessarily equivalent to `x <= y;`. The sequential block

```
initial begin
   x <= 1;
   x <= 0;
end
```

produces a guaranteed sequence of values for `x`, but

```
initial begin
   x <= #0 1;
   x <= #0 0;
end
```

has an indeterminate result. There is nothing explicitly said about this last point in the *Standard*, but it is the natural interpretation, given the above.

The delayed nonblocking update events are referred to as the *future nonblocking assignment update events* in section 5.3 of the *Standard*. We can see that this is something of a misnomer, however, and the nondeterministic aspect of such events can be more clearly seen if we dispense entirely with this term. Instead, we adopt the interpretation that in a nonblocking assignment with intra-assignment control, it is not a *nonblocking update* that is scheduled but rather a new *delay-blocked* thread, whose body consists of a *nonblocking assignment statement*.

Now the behavior is obvious. Ordinary nonblocking assignments schedule their updates to occur "in the order the statements were executed" ([12], p. 47). Since an intra-assignment timing control actually causes the creation of a new *thread*, which is then blocked, and since all threads scheduled to run at a certain time do so in arbitrary order, we immediately see the resulting nondeterminism.

Event Controls

Of all the constructs in our semantics, event controls pose perhaps the greatest challenge to effective representation. There are many reasons for this, but the main one seems to be the fact that what is guarded upon—the rising or falling edge of a change in signal level—is an event, not a data value, and consequently, the ordinary means for representing the memory state are insufficient. This is the impetus for the double representation of the store, which allows us at least an indirect way to express these edges, but the resulting axioms are fairly complicated.

Overall, we are concerned with accurately expressing three aspects of event control behavior: blocking on an event control in an *active* thread, release by a guard (also called *firing*) when the relevant event occurs, and the removal of these events from the computational state when no more guards can fire on them.

To define evaluation order, our grammar includes an evaluation context for guards. Again, we choose left-to-right ordering for the sake of simplicity only:

$$\begin{array}{rcl} \mathcal{G} & ::= & [\,] \\ & | & \mathcal{G}\ \texttt{or}\ \langle guard \rangle \\ & | & \texttt{posedge}\ \langle \mathsf{ID} \rangle\ \texttt{or}\ \mathcal{G} \\ & | & \texttt{negedge}\ \langle \mathsf{ID} \rangle\ \texttt{or}\ \mathcal{G} \end{array}$$

Unlike the evaluation context for expressions, $\mathcal{E}$, we do not define any variable lookup on $\mathcal{G}$, since events hold no data.

The specification of blocking itself is the most straightforward task, since we do not to be concerned with the type of event control used:

$$\begin{array}{l} \{\mathsf{AT} \parallel \mathsf{THREAD}\ [\ \langle event\_control \rangle\ S\ ]\}\ \ \mathsf{FG} \\ \quad \longrightarrow\ \ \mathsf{AT}\ \ \{\mathsf{FG} \parallel \mathsf{THREAD}\ [\ \langle event\_control \rangle\ S\ ]\} \end{array}$$

This expresses correctly the form of the reduction itself: we simply rewrite the representation of the thread so that it is no longer an *active thread*. It is not entirely accurate, however, because it allows an event control to block while the event on which it guards is present in memory. As a result, we could fire a guard on an event that occurred *before* the event control blocked.

Only event controls that have already blocked their threads can see an edge, however, and a failure to understand this can lead to incorrect analysis of a program execution. Consider:

```
initial begin
   x = 0;
   x = 1;
end

always @(posedge x) some_statement
```

In our representation, these are both *active* threads:

```
initial begin #0#0#0
   x = 0;
   x = 1;
end

initial begin #0#0#0
  forever @(posedge x) some_statement
end
```

Whether the `posedge x` guard executes depends entirely on the order in which a particular simulator evaluates the source code[4]. From a formal point of view, the execution is indeterminate. In order, therefore, to prevent a guard in an active thread from incorrectly releasing its thread, we will keep the reduction as written, but require that before it can be applied, the two copies of the store are identical (*i.e.*, no edges are present). This is slightly more conservative than we need—the real requirement is that there be no edges on any variable occurring within the event control—but this form is simpler to write:

[4]As written, the Veriwell simulator will fire the guard—if we reverse the order of the threads, it will not.

(28) $\mathsf{ST}_1\ \mathsf{ST}_2\ \{\mathsf{AT} \parallel \mathsf{THREAD}\ [\ \langle \mathit{event\_control} \rangle\ S\ ]\}\ \mathsf{FG}$
$\longrightarrow\ \mathsf{ST}_1\ \mathsf{ST}_2\ \mathsf{AT}\ \{\mathsf{FG} \parallel \mathsf{THREAD}\ [\ \langle \mathit{event\_control} \rangle\ S\ ]\}$
$\bullet$ if $\mathsf{ST}_1 = \mathsf{ST}_2$

In order to correctly define guard *release*, we need to consider a number of facts about its behavior. The first is the manner in which a guard releases its thread. Gordon ([9], pp.4 and 10) presents results that suggest strongly that a guarded statement is actually released with a zero-delay, so that it executes strictly after all the threads that were scheduled to run when the release occurred. We shall assume this convention here, as well.

The second fact is a point about the evaluation of the guard itself; namely that, if the value on which a guard is watching is greater than one bit, the evaluation always takes place on the *low-order bit* ([12], p.114). The four-values allowed on each bit make the rules for firing a guard fairly complicated, and like most of the rules for expression evaluation, they are tedious to express in a formal manner. On the other hand, no semantic description of guard release would be complete without them, so we will introduce additional notation here, $lsb()$, which indicates the *least-significant bit* of its argument.

Finally, when an edge occurs, *every* thread guarding on that event must fire: "When an update event is executed, all the processes that are sensitive to that event are evaluated in an arbitrary order" ([12], p.45). As with assignment, therefore, we must take care to leave an edge alone and not alter the memory state in any way when a guard fires.

The result is a set of three similar axioms, accounting for `posedge`, `negedge`, and generic guard release:

(29) $\mathsf{ST}_1[x = n_1]\ \mathsf{ST}_2[x = n_2]\ \mathsf{AT}\ \ \{\mathsf{FG} \,\|\, \mathsf{THREAD}\ [\ \mathtt{@}(\mathcal{G}[\mathtt{posedge}\ x])\ S\ ]\}$
$\quad\longrightarrow\ \mathsf{ST}_1[x = n_1]\ \mathsf{ST}_2[x = n_2]\ \ \{\mathsf{AT} \,\|\, \mathsf{THREAD}\ [\ \mathtt{\#0}\ S\ ]\}\ \ \mathsf{FG}$
- if $lsb(n_1) \neq 1$ and $lsb(n_2) = 1$, *or*
- if $lsb(n_1) = 0$ and $lsb(n_2) \neq 0$

(30) $\mathsf{ST}_1[x = n_1]\ \mathsf{ST}_2[x = n_2]\ \mathsf{AT}\ \ \{\mathsf{FG} \,\|\, \mathsf{THREAD}\ [\ \mathtt{@}(\mathcal{G}[\mathtt{negedge}\ x])\ S\ ]\}$
$\quad\longrightarrow\ \mathsf{ST}_1[x = n_1]\ \mathsf{ST}_2[x = n_2]\ \ \{\mathsf{AT} \,\|\, \mathsf{THREAD}\ [\ \mathtt{\#0}\ S\ ]\}\ \ \mathsf{FG}$
- if $lsb(n_1) \neq 0$ and $lsb(n_2) = 0$, *or*
- if $lsb(n_1) = 1$ and $lsb(n_2) \neq 1$

(31) $\mathsf{ST}_1[x = n_1]\ \mathsf{ST}_2[x = n_2]\ \mathsf{AT}\ \ \{\mathsf{FG} \,\|\, \mathsf{THREAD}\ [\ \mathtt{@}(\mathcal{G}[x])\ S\ ]\}$
$\quad\longrightarrow\ \mathsf{ST}_1[x = n_1]\ \mathsf{ST}_2[x = n_2]\ \ \{\mathsf{AT} \,\|\, \mathsf{THREAD}\ [\ \mathtt{\#0}\ S\ ]\}\ \ \mathsf{FG}$
- if $lsb(n_1) \neq lsb(n_2)$

We are left, finally, with the question of when the representation of an edge should be removed from a program. We have already stated the rather straightforward requirement that every guard should have the opportunity to fire when the relevant event occurs. But is every blocked guard *guaranteed* to fire? Although the *Standard* does not say so explicitly, an affirmative answer again seems natural. Thus in our semantics, a blocked guard *is* guaranteed to fire when its associated event occurs.

The immediate consequence of this is that we must impose some complicated side conditions on the axioms that define edge removal, which say, in effect, that there are no more guards left to fire on this particular edge:

(32) $\mathsf{ST}_1[x = n_1]\ \mathsf{ST}_2[x = n_2]\ \mathsf{FG} \longrightarrow \mathsf{ST}_1[x = n_2]\ \mathsf{ST}_2[x = n_2]\ \mathsf{FG}$
- if both of the following hold :
  - ∘ either $(lsb(n_1) \neq 1$ and $lsb(n_2) = 1)$ or $(lsb(n_1) = 0$ and $lsb(n_2) \neq 0)$
  - ∘ there is no $t \parallel \mathsf{FG}$ such that either :
    - $\diamond t =$ THREAD [@($\mathcal{G}[x]$) S], or
    - $\diamond t =$ THREAD [@($\mathcal{G}[$posedge$x]$) S]

(33) $\mathsf{ST}_1[x = n_1]\ \mathsf{ST}_2[x = n_2]\ \mathsf{FG} \longrightarrow \mathsf{ST}_1[x = n_2]\ \mathsf{ST}_2[x = n_2]\ \mathsf{FG}$
- if both of the following hold :
  - ∘ either $(lsb(n_1) \neq 0$ and $lsb(n_2) = 0)$ or $(lsb(n_1) = 1$ and $lsb(n_2) \neq 1)$
  - ∘ there is no $t \parallel \mathsf{FG}$ such that either :
    - $\diamond t =$ THREAD [@($\mathcal{G}[x]$) S], or
    - $\diamond t =$ THREAD [@($\mathcal{G}[$negedge$x]$) S]

The exception to this is the case where an edge was not really present in the first place. Since edge-detection only occurs on the low order bit, this amounts to an assertion that we can immediately update the value of a variable from ST_2 to ST_1, if the two values have the same lsb:

(34) $\mathsf{ST}_1[x = n_1]\ \mathsf{ST}_2[x = n_2] \longrightarrow \mathsf{ST}_1[x = n_2]\ \mathsf{ST}_2[x = n_2]$
- if $lsb(n_1) = lsb(n_2)$

Finally, we address a point that may have been overlooked by the reader up to now: the fact that when we chose our subset of Verilog, we limited a guard only to simple identifiers. The full Verilog grammar, in fact, allows any expression whatsoever to be used. However, we have purposefully chosen to limit guards, for two reasons.

The first is aesthetic: allowing arbitrary expressions would vastly complicate the rules for determining whether or not an event had actually occurred at a given point in the program, without providing much in return in the way of useful semantic knowledge.

The second point is more serious, however: if we allow the evaluation of arbitrary expressions within guards, then guard release becomes susceptible to the same interleaving problems suffered by expression evaluation. As our semantics stands, we are able to guarantee that when an event occurs, every guard that has blocked on that event will fire. Since guards only watch on simple identifiers, this guarantee is independent of the interleaving problem in expression evaluation, which our semantics allows. Were we to allow an arbitrary expression in a guard, we would lose this guarantee. Under the present semantics, which we believe are implied by the *Standard*, there is the possibility of a state change during expression evaluation, which could result in an edge on the value of some expression that disappears before the expression can be evaluated.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

In the present work, we have defined a formal semantics for a large subset of behavioral Verilog that works entirely by transformations from Verilog programs to Verilog programs. As a result, our semantics can be used to reason about Verilog programs in a *direct* fashion. One of the key features of this approach is that we have been able to represent in a formal manner not only the memory state, but also the also the state of the discrete event queue whose behavior defines the semantics of Verilog constructs.

However, there are clearly areas for future improvement.

First, the semantics should be expanded to account for the full behavioral subset of Verilog. We have ommitted several often-used constructs from our subset, largely on the grounds of their redundancy. Most of them—`for`, `repeat`, `case`, etc.—could be easily included with no real modifications to the existing axioms. Others, such as `wait`, `disable` and `fork/join` would probably some minor changes to the overall structure of many axioms, particularly with respect to some of the side conditions.

The introduction of `function` poses a special challenge, since the assumption that expressions are free of side-effects would no longer hold. Boehm [2] proposes a way to axiomatize side-effects and aliasing under this more general condition.

Second, while the semantics given in this thesis are accurate models of Verilog behavior, there are probably alternative formulations that are just as correct, but produce better code from the point of view of their synthesizability.

Can we find transformations that take behavioral programs to equivalent structural ones? This question is still open, but developing such a set of transformations would be tantamount to developing a method for behavioral synthesis by program transformation. This correlation with synthesis suggests that one fruitful approach might be to investigate to what degree we could apply the axiomatic approach toward the deduction of certain data and control-flow properties. Manual translation of Verilog programs from behavioral to structural versions usually results in a set of modules that represent in a direct fashion the control and data-flow properties of the original behavioral module. Further, the extraction of data and control-flow graphs forms the basis of many of the successful results in automatic synthesis ([3], [1], ([5]).

All of these works sharply restrict the syntax of behavioral constructs and limit the set of allowable programs to those written in styles known to be synthesizable. It seems certain that we would have to do the same. The formal semantics described in this thesis imposes many fewer restrictions on syntax and program structure, and as a result, we have had to trade off synthesizability in our axioms for the sake of more general behavioral descriptions.

Finally, the greatest limitation of the present semantics is, of course, the exclusion of the structural subset of Verilog. The successful application of the axiomatic approach to a formal representation of structural Verilog remains an open problem, as does the integration of the behavioral and structural constructs in a unified semantics.

In many ways, continuous and procedural assignment are unrelated, which makes representation of one in terms of the other extremely problematic. Only a few easy direct translations are possible between the two, since the register and net data types on which each operates cannot be interchanged. There are behavioral aspects of wire connections that have no analogue at all within the behavioral subset—*inertial delay*, for instance ([9], p.5), or feedback (connection of a gate to itself). The reverse is, of course, also true. Further, many of the semantics of the structural constructs' timing properties do not appear to be entirely well-understood. There is ongoing work by members of the Cambridge VFE project to identify this part of the semantics ([8], [10], [15], [16]).

It is likely that the integration of the structural constructs into our semantics will require some changes to the current approach. Inclusion of these constructs means that we will have to keep track of two more aspects of the computational state: the values on the sinks of each wire, and the next value (if it is different) that is scheduled to appear at the sink, along with the time of its appearance. Obviously, we cannot present this as a series of simple assignments, since procedural assignment does not apply to net data types. At this time, we are working on alternative approaches.

APPENDIX A

SYNTAX

In the following extended BNF grammar, a number of notational conventions are observed. We use the common BNF characters "::=" and "|" to represent "is defined as" and "or", respectively. The special symbol "ϵ" is used to represent the null token. The symbols "$*$" and "?", when they immediately follow terminal or non-terminal items, represent (respectively) Kleene-closure and "zero or one occurrences" of that item[1].

With the exception of these five special symbols, all characters outside of those contained within angle brackets occur as literals within the source code. Words in angle brackets represent terminal and non-terminal symbols: neither the angle brackets nor the chararacters inside of them appear literally in a program. Terminal tokens are represented within angle brackets as capitalized words. All other words within angle brackets represent non-terminals.

[1]The reader should take care to distinguish between the BNF symbol "?" and the keyword character "?", which appears in "$\langle exp \rangle$? $\langle exp \rangle$: $\langle exp \rangle$"

Syntax of Behavioral Verilog Subset

$\langle \mathit{program} \rangle$	$::=$	$\langle \mathit{module} \rangle *$
$\langle \mathit{module} \rangle$	$::=$	`module` $\langle \mathsf{MODULE\_ID} \rangle$ $\langle \mathit{port\_list} \rangle$? `;`
		$\langle \mathit{set\_of\_declarations} \rangle$
		$\langle \mathit{module\_body} \rangle$
		`endmodule`
$\langle \mathit{port\_list} \rangle$	$::=$	`(`$\langle \mathit{port} \rangle$ $\langle$`,`$\langle \mathit{port} \rangle\rangle *$`)`
$\langle \mathit{port} \rangle$	$::=$	$\langle \mathsf{PORT\_ID} \rangle$
$\langle \mathit{module\_body} \rangle$	$::=$	$\langle \mathit{thread} \rangle *$
$\langle \mathit{set\_of\_declarations} \rangle$	$\mid$	$\langle \mathit{declaration} \rangle *$
$\langle \mathit{declaration} \rangle$	$::=$	$\langle \mathit{reg\_decl} \rangle \mid \langle \mathit{net\_decl} \rangle$
	$\mid$	$\langle \mathit{input\_decl} \rangle \mid \langle \mathit{output\_decl} \rangle$
$\langle \mathit{reg\_decl} \rangle$	$::=$	`reg` $\langle \mathsf{REG\_ID} \rangle$ $\langle$`,`$\langle \mathsf{REG\_ID} \rangle\rangle *$`;`
$\langle \mathit{net\_decl} \rangle$	$::=$	`wire` $\langle \mathsf{NET\_ID} \rangle$ $\langle$`,`$\langle \mathsf{NET\_ID} \rangle\rangle *$`;`
$\langle \mathit{input\_decl} \rangle$	$::=$	`input` $\langle \mathsf{PORT\_ID} \rangle$ $\langle$`,`$\langle \mathsf{PORT\_ID} \rangle\rangle *$`;`
$\langle \mathit{output\_decl} \rangle$	$::=$	`output` $\langle \mathsf{PORT\_ID} \rangle$ $\langle$`,`$\langle \mathsf{PORT\_ID} \rangle\rangle *$`;`
$\langle \mathit{thread} \rangle$	$::=$	$\langle \mathit{always\_construct} \rangle$
	$\mid$	$\langle \mathit{initial\_construct} \rangle$
$\langle \mathit{always\_construct} \rangle$	$::=$	`always` $\langle \mathit{stat} \rangle$
$\langle \mathit{initial\_construct} \rangle$	$::=$	`initial` $\langle \mathit{stat} \rangle$
$\langle \mathit{stat} \rangle$	$::=$	ϵ`;`
	$\mid$	$\langle \mathit{untimed\_stat} \rangle$
	$\mid$	$\langle \mathit{timed\_stat} \rangle$
$\langle \mathit{untimed\_stat} \rangle$	$::=$	$\langle \mathit{blocking\_asgn} \rangle$
	$\mid$	$\langle \mathit{nonblocking\_asgn} \rangle$
	$\mid$	`if (` $\langle \mathit{exp} \rangle$ `)` $\langle \mathit{stat} \rangle$
	$\mid$	`if (` $\langle \mathit{exp} \rangle$ `)` $\langle \mathit{stat} \rangle$ `else` $\langle \mathit{stat} \rangle$
	$\mid$	`forever` $\langle \mathit{stat} \rangle$
	$\mid$	`while (`$\langle \mathit{exp} \rangle$`)` $\langle \mathit{stat} \rangle$
	$\mid$	$\langle \mathit{seq\_block} \rangle$
$\langle \mathit{timed\_stat} \rangle$	$::=$	$\langle \mathit{delay\_or\_event\_control} \rangle$ $\langle \mathit{stat} \rangle$
$\langle \mathit{blocking\_asgn} \rangle$	$::=$	$\langle \mathit{reg\_lvalue} \rangle$ `=` $\langle \mathit{delay\_or\_event\_control} \rangle$? $\langle \mathit{exp} \rangle$`;`
$\langle \mathit{nonblocking\_asgn} \rangle$	$::=$	$\langle \mathit{reg\_lvalue} \rangle$ `<=` $\langle \mathit{delay\_or\_event\_control} \rangle$? $\langle \mathit{exp} \rangle$`;`
$\langle \mathit{seq\_block} \rangle$	$::=$	`begin` $\langle \mathit{statlist} \rangle$ `end`
$\langle \mathit{statlist} \rangle$	$::=$	ϵ
	$\mid$	$\langle \mathit{stat} \rangle$ $\langle \mathit{statlist} \rangle$
$\langle \mathit{delay\_or\_event\_control} \rangle$	$::=$	$\langle \mathit{delay\_control} \rangle$
	$\mid$	$\langle \mathit{event\_control} \rangle$

```
⟨delay_control⟩   ::=  #⟨exp⟩
⟨event_control⟩   ::=  @(⟨guard⟩)
⟨reg_lvalue⟩      ::=  ⟨REG_ID⟩
⟨guard⟩           ::=  ⟨ID⟩
                    |  posedge ⟨ID⟩
                    |  negedge ⟨ID⟩
                    |  ⟨guard⟩ or ⟨guard⟩
⟨exp⟩             ::=  ⟨ID⟩ | ⟨NUMBER⟩
                    |  ⟨u_op⟩⟨exp⟩
                    |  ⟨exp⟩ ⟨b_op⟩ ⟨exp⟩
                    |  ⟨exp⟩? ⟨exp⟩: ⟨exp⟩
⟨u_op⟩            ::=  + | - | ! | ~ | & | ~& | | | ~| | ^ | ~^ | ^~
⟨b_op⟩            ::=  + | - | * | / | % | == | != | === | !== | && | ||
                         | < | <= | > | >= | & | | | |^ | ^~ | ~^ | << | >>
```

<u>Evaluation Contexts</u>

```
ℰ                 ::=  [ ]
                    |  ⟨u_op⟩ ℰ
                    |  ℰ ⟨b_op⟩ ⟨exp⟩
                    |  ⟨NUMBER⟩ ⟨b_op⟩ ℰ
                    |  ℰ? ⟨exp⟩: ⟨exp⟩
𝒢                 ::=  [ ]
                    |  𝒢 or ⟨guard⟩
                    |  posedge ⟨ID⟩ or 𝒢
                    |  negedge ⟨ID⟩ or 𝒢
```

APPENDIX B

AXIOMATIC DESCRIPTION OF VERILOG

See Chapter IV for an explanation of the notation used.

Ordinary Statements

(1) $\texttt{if (} n \texttt{) } S \longrightarrow S$ $\quad\bullet$ if $n \neq 0$

(2) $\texttt{if (0) } S \longrightarrow \epsilon$

(3) $\texttt{if (} n \texttt{) } S_1 \texttt{ else } S_2 \longrightarrow S_1$ $\quad\bullet$ if $n \neq 0$

(4) $\texttt{if (0) } S_1 \texttt{ else } S_2 \longrightarrow S_2$

(5) $\texttt{while (}\langle exp\rangle\texttt{) } S$

$\quad\longrightarrow \texttt{if (}\langle exp\rangle\texttt{) begin } S\texttt{; while (}\langle exp\rangle\texttt{) } S \texttt{ end}$

(6) $\texttt{forever } S \longrightarrow \texttt{while (1) } S$

(7) $\epsilon\texttt{; } S \longrightarrow S$

(8) $\mathsf{THREAD}\ [\ \texttt{begin } S\texttt{; } \langle statlist\rangle \texttt{ end}\ ]$

$\quad\longrightarrow \mathsf{THREAD}\ [\ S\texttt{; begin } \langle statlist\rangle \texttt{ end}\ ]$

(9) $\mathsf{THREAD}\ [\ \texttt{begin } \epsilon \texttt{ end } S\ ]$

$\quad\longrightarrow \mathsf{THREAD}\ [\ S\ ]$

(10) $\mathsf{THREAD}\ [\ \epsilon\ ] \longrightarrow \epsilon$

Access to the Store (Excluding Event Controls)

(11) $\mathsf{ST}_2[x = n]\ \mathsf{AT}[\texttt{if (} \mathcal{E}[x] \texttt{) } S]$

$\quad\longrightarrow \mathsf{ST}_2[x = n]\mathsf{AT}[\texttt{if (} \mathcal{E}[n] \texttt{) } S]$

$$(12)\quad \mathsf{ST}_2[x \texttt{ = } n]\ \mathsf{AT}[\texttt{if (} \mathcal{E}[x] \texttt{) } S_1 \texttt{ else } S_2] \\ \longrightarrow\ \mathsf{ST}_2[x \texttt{ = } n]\ \mathsf{AT}[\texttt{if (} \mathcal{E}[n] \texttt{) } S_1 \texttt{ else } S_2]$$

$$(13)\quad \mathsf{ST}_2[x_1 \texttt{ = } n]\ \mathsf{AT}[x_2 \texttt{ = } \mathcal{E}[x_1]] \\ \longrightarrow\ \mathsf{ST}_2[x_1 \texttt{ = } n]\ \mathsf{AT}[x_2 \texttt{ = } \mathcal{E}[n]]$$

$$(14)\quad \mathsf{ST}_2[x_1 \texttt{ = } n]\ \mathsf{AT}[x_2 \texttt{ = } \langle \mathit{delay\_or\_event\_control} \rangle\ \mathcal{E}[x_1]] \\ \longrightarrow\ \mathsf{ST}_2[x_1 \texttt{ = } n]\ \mathsf{AT}[x_2 \texttt{ = } \langle \mathit{delay\_or\_event\_control} \rangle\ \mathcal{E}[n]]$$

$$(15)\quad \mathsf{ST}_2[x_1 \texttt{ = } n]\ \mathsf{AT}[x_2 \texttt{ = \#}\mathcal{E}[x_1]\ \langle \mathit{exp} \rangle] \\ \longrightarrow\ \mathsf{ST}_2[x_1 \texttt{ = } n]\ \mathsf{AT}[x_2 \texttt{ = \#}\mathcal{E}[n]\ \langle \mathit{exp} \rangle]$$

$$(16)\quad \mathsf{ST}_2[x_1 \texttt{ = } n]\ \mathsf{AT}[x_2 \texttt{ <= } \mathcal{E}[x_1]] \\ \longrightarrow\ \mathsf{ST}_2[x_1 \texttt{ = } n]\ \mathsf{AT}[x_2 \texttt{ <= } \mathcal{E}[n]]$$

$$(17)\quad \mathsf{ST}_2[x_1 \texttt{ = } n]\ \mathsf{AT}[x_2 \texttt{ <= } \langle \mathit{delay\_or\_event\_control} \rangle\ \mathcal{E}[x_1]] \\ \longrightarrow\ \mathsf{ST}_2[x_1 \texttt{ = } n]\ \mathsf{AT}[x_2 \texttt{ <= } \langle \mathit{delay\_or\_event\_control} \rangle\ \mathcal{E}[n]]$$

$$(18)\quad \mathsf{ST}_2[x_1 \texttt{ = } n]\ \mathsf{AT}[x_2 \texttt{ <= \#}\mathcal{E}[x_1]\ \langle \mathit{exp} \rangle] \\ \longrightarrow\ \mathsf{ST}_2[x_1 \texttt{ = } n]\ \mathsf{AT}[x_2 \texttt{ <= \#}\mathcal{E}[n]\ \langle \mathit{exp} \rangle]$$

$$(19)\quad \mathsf{ST}_2[x \texttt{ = } n]\ \mathsf{AT}[\texttt{\#}\mathcal{E}[x]\ S] \\ \longrightarrow\ \mathsf{ST}_2[x \texttt{ = } n]\ \mathsf{AT}[\texttt{\#}\mathcal{E}[n]\ S]$$

$$(20)\quad \mathsf{ST}_1[x \texttt{ = } n_1]\ \mathsf{ST}_2[x \texttt{ = } n_1]\ \mathsf{AT}[x \texttt{ = } n_2\texttt{;}\ S] \quad \longrightarrow \quad \mathsf{ST}_1[x \texttt{ = } n_1]\ \mathsf{ST}_2[x \texttt{ = } n_2]\ \mathsf{AT}[S]$$

<u>Non-Blocking Assignment</u>

$$(21)\quad \mathsf{AT}[x \texttt{ <= } n\texttt{;}\ S]\ \{\texttt{initial begin: \_NBU}\ \mathsf{NBUPDATES}_i\ \texttt{end}\} \\ \longrightarrow\ \mathsf{AT}[S]\ \{\texttt{initial begin: \_NBU}\ \mathsf{NBUPDATES}_i\ \mathrm{x}\ \texttt{<=}\ \mathrm{n}\texttt{;end}\}$$

(22) $\mathsf{ST}_1\ \mathsf{ST}_2\ \mathsf{AT}\ \mathsf{DT}\ \{\texttt{initial begin: \_NBU}\ \mathsf{NBUPDATES}_i\ \texttt{end}\}$
$\longrightarrow\ \mathsf{ST}_1\ \mathsf{ST}_2\ \{\mathsf{THREAD}\ [\ \mathsf{UPDATES}_i\ ] \,\|\, \mathsf{AT}\}\ \mathsf{DT}$
$\texttt{initial begin: \_NBU}\ \epsilon\ \texttt{end}$
- if $\mathsf{AT} = \mathsf{DT} = \epsilon$, and $\mathsf{ST}_1 = \mathsf{ST}_2$
- each nonblocking assignment in $\mathsf{NBUPDATES}_\mathrm{i}$ is transformed to a blocking assignment in $\mathsf{UPDATES}_\mathrm{i}$, according to the schema :
  x `<=` n ; $\longrightarrow$ x `=` n ;

Scheduling of Delayed Threads

(23) $\{\mathsf{THREAD}\ [\ \texttt{\#}n\ S\ ] \,\|\, \mathsf{AT}\}\ \mathsf{FD}\ \longrightarrow\ \mathsf{AT}\ \{\mathsf{FD} \,\|\, \mathsf{THREAD}\ [\ \texttt{\#}n\ S\ ]\}$
- if $n > 1$

(24) $\mathsf{ST}_1\ \mathsf{ST}_2\ \mathsf{AT}\ \mathsf{DT}\ \mathsf{NBU}\ \{\mathsf{FD} \,\|\, \mathsf{THREADLIST}_i\}$
$\longrightarrow\ \mathsf{ST}_1\ \mathsf{ST}_2\ \{\mathsf{THREADLIST}_i \,\|\, \mathsf{AT}\}\ \mathsf{DT}\ \mathsf{NBU}\ \mathsf{FD}$
- if $\mathsf{AT} = \mathsf{DT} = \mathsf{NBU} = \mathsf{FD} = \epsilon$, and $\mathsf{ST}_1 = \mathsf{ST}_2$
- the body of each THREAD in $\mathsf{THREADLIST}_i$ is transformed to an *active thread* representation, according to the schema :
  $\mathsf{THREAD}\ [\ \texttt{\#}n\ S\ ]\ \longrightarrow\ \mathsf{THREAD}\ [\ \texttt{\#(}n\texttt{-1)}\ S\ ]$, *if* $n > 1$
  $\mathsf{THREAD}\ [\ \texttt{\#1}\ S\ ]\ \longrightarrow\ \mathsf{THREAD}\ [\ S\ ]$, *otherwise*

(25) $\mathsf{ST}_1\ \mathsf{ST}_2\ \mathsf{AT}\ \{\mathsf{DT} \,\|\, \mathsf{THREADLIST}_i\}$
$\longrightarrow\ \mathsf{ST}_1\ \mathsf{ST}_2\ \{\mathsf{THREADLIST}_i \,\|\, \mathsf{AT}\}\ \mathsf{DT}$
- if $\mathsf{AT} = \mathsf{DT} = \epsilon$, and $\mathsf{ST}_1 = \mathsf{ST}_2$
- each THREAD in $\mathsf{THREADLIST}_\mathrm{i}$ is transformed to an *active thread* representation, according to the schema :
  `initial begin #0#0#0#0` S `end`
  $\longrightarrow$ `initial begin #0#0#0` S `end`

(26) x `=` $\langle \textit{delay\_or\_event\_control} \rangle\ n$; $\longrightarrow$ $\langle \textit{delay\_or\_event\_control} \rangle\ x$ `=` n ;

(27) $\mathsf{AT}[x$ `<=` $\langle \textit{delay\_or\_event\_control} \rangle\ n\ ;\ S]$
$\longrightarrow\ \{\mathsf{AT}[S] \,\|\, \mathsf{THREAD}\ [\ \langle \textit{delay\_or\_event\_control} \rangle\ x$ `<=` $n\ ;\ ]\}$

Event Controls

(28) $\mathsf{ST}_1\ \mathsf{ST}_2\ \{\mathsf{AT} \parallel \mathsf{THREAD}\ [\ \langle event\_control \rangle\ S\ ]\}\ \mathsf{FG}$
$\longrightarrow\ \mathsf{ST}_1\ \mathsf{ST}_2\ \mathsf{AT}\ \{\mathsf{FG} \parallel \mathsf{THREAD}\ [\ \langle event\_control \rangle\ S\ ]\}$
• if $\mathsf{ST}_1 = \mathsf{ST}_2$

(29) $\mathsf{ST}_1[x = n_1]\ \mathsf{ST}_2[x = n_2]\ \mathsf{AT}\ \{\mathsf{FG} \parallel \mathsf{THREAD}\ [\ \texttt{@}(\mathcal{G}[\texttt{posedge}\ x])\ S\ ]\}$
$\longrightarrow\ \mathsf{ST}_1[x = n_1]\ \mathsf{ST}_2[x = n_2]\ \{\mathsf{AT} \parallel \mathsf{THREAD}\ [\ \texttt{\#}0\ S\ ]\}\ \mathsf{FG}$
• if $lsb(n_1) \neq 1$ and $lsb(n_2) = 1$, *or*
• if $lsb(n_1) = 0$ and $lsb(n_2) \neq 0$

(30) $\mathsf{ST}_1[x = n_1]\ \mathsf{ST}_2[x = n_2]\ \mathsf{AT}\ \{\mathsf{FG} \parallel \mathsf{THREAD}\ [\ \texttt{@}(\mathcal{G}[\texttt{negedge}\ x])\ S\ ]\}$
$\longrightarrow\ \mathsf{ST}_1[x = n_1]\ \mathsf{ST}_2[x = n_2]\ \{\mathsf{AT} \parallel \mathsf{THREAD}\ [\ \texttt{\#}0\ S\ ]\}\ \mathsf{FG}$
• if $lsb(n_1) \neq 0$ and $lsb(n_2) = 0$, *or*
• if $lsb(n_1) = 1$ and $lsb(n_2) \neq 1$

(31) $\mathsf{ST}_1[x = n_1]\ \mathsf{ST}_2[x = n_2]\ \mathsf{AT}\ \{\mathsf{FG} \parallel \mathsf{THREAD}\ [\ \texttt{@}(\mathcal{G}[x])\ S\ ]\}$
$\longrightarrow\ \mathsf{ST}_1[x = n_1]\ \mathsf{ST}_2[x = n_2]\ \{\mathsf{AT} \parallel \mathsf{THREAD}\ [\ \texttt{\#}0\ S\ ]\}\ \mathsf{FG}$
• if $lsb(n_1) \neq lsb(n_2)$

(32) $\mathsf{ST}_1[x = n_1]\ \mathsf{ST}_2[x = n_2]\ \mathsf{FG}\ \longrightarrow\ \mathsf{ST}_1[x = n_2]\ \mathsf{ST}_2[x = n_2]\ \mathsf{FG}$
• if both of the following hold :
∘ either $(lsb(n_1) \neq 1$ and $lsb(n_2) = 1)$ or $(lsb(n_1) = 0$ and $lsb(n_2) \neq 0)$
∘ there is no $t \parallel \mathsf{FG}$ such that either :
⋄ $t =\ \mathsf{THREAD}\ [\ \texttt{@}(\mathcal{G}[x])\ S\ ]$, or
⋄ $t =\ \mathsf{THREAD}\ [\ \texttt{@}(\mathcal{G}[\texttt{posedge}x])\ S\ ]$

(33) $\mathsf{ST}_1[x = n_1]\ \mathsf{ST}_2[x = n_2]\ \mathsf{FG}\ \longrightarrow\ \mathsf{ST}_1[x = n_2]\ \mathsf{ST}_2[x = n_2]\ \mathsf{FG}$
• if both of the following hold :
∘ either $(lsb(n_1) \neq 0$ and $lsb(n_2) = 0)$ or $(lsb(n_1) = 1$ and $lsb(n_2) \neq 1)$
∘ there is no $t \parallel \mathsf{FG}$ such that either :
⋄ $t =\ \mathsf{THREAD}\ [\ \texttt{@}(\mathcal{G}[x])\ S\ ]$, or
⋄ $t =\ \mathsf{THREAD}\ [\ \texttt{@}(\mathcal{G}[\texttt{negedge}x])\ S\ ]$

(34) $\mathsf{ST}_1[x = n_1]\ \mathsf{ST}_2[x = n_2]\ \longrightarrow\ \mathsf{ST}_1[x = n_2]\ \mathsf{ST}_2[x = n_2]$
• if $lsb(n_1) = lsb(n_2)$

BIBLIOGRAPHY

[1] M.G. Arnold and J.D. Shuler. A Synthesis Preprocessor that Converts Implicit Style Verilog into One-Hot Designs. In *1997 IEEE International Verilog HDL Conference*, pages 38–45, Washington, 1997. IEEE Press.

[2] H. Boehm. Side Effects and Aliasing Can Have Simple Axiomatic Descriptions. *ACM Transactions on Programming Languages and Systems*, 7(4):637–655, October 1985.

[3] R. Camposano. Behavior-Preserving Transformations for High-Level Synthesis. In *Hardware Specification, Verification, and Synthesis: Mathematical Aspects*, pages 106–128. Springer-Verlag, New York, 1990.

[4] S. Cheng. Compilation, Synthesis, and Simulation of Hardware Description Languages—The Compositional Models of HDL's. Phd thesis, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1998.

[5] S. Cheng and R. Brayton. Compiling Verilog into Automata. Memorandum UCB/ERC M94/37, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1994.

[6] S. Cheng and R. Brayton. Synthesizing Multi-Phase HDL Programs. In *1996 IEEE International Verilog HDL Conference*, pages 67–76, Washington, 1996. IEEE Press.

[7] S. Cheng, R. Brayton, G. York, K. Yelick, and A. Saldanha. Compiling Verilog into Timed Finite State Machines. In *1995 IEEE International Verilog HDL Conference*, pages 32–39, Washington, 1995. IEEE Press.

[8] M.J.C. Gordon. Event and Cycle Semantics of Hardware Description Languages. Unpublished draft, available (as of 08-24-98) on the World-Wide Web at `http://www.cl.cam.uk/users/mjcg/Verilog/V/HDLPaper.ps`.

[9] M.J.C. Gordon. The Semantic Challenge of Verilog HDL. In *Tenth Annual IEEE Symposium on Logic in Computer Science (LICS '95)*, June 1995. Revised version available (as of 08-24-98) on the World-Wide Web at `http://www.cl.cam.uk/users/mjcg/Verilog/V.ps.Z`.

[10] M.J.C. Gordon and A. Ghosh. Language Independent RTL Semantics. In *1998 IEEE CS Annual Workshop on VLSI: System Level Design*, to be published. Available (as of 08-24-98) on the World-Wide Web at `http://www.cl.cam.uk/users/mjcg/OrlandoPaper.ps`.

[11] X. Hua and H. Zang. Axiomatic Semantics of a Hardware Specification Language. In *Second Great Lakes Symposium on VLSI*, pages 183–190, Los Alamitos, CA, 1991. IEEE Press.

[12] IEEE. *Standard Hardware Description Language Based on the Verilog Hardware Description Language*. IEEE Press, Los Alamitos, CA, 1996. IEEE Standard 1364-1995.

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

[14] A. Sabry and J. Field. Reasoning About Explicit and Implicit Representations of State. In *ACM SIGPLAN Workshop on State in Programming Languages*, pages 17–30, 1993. Tech. Rep. YALEU/DCS/RR-968, Dept. of Computer Science, Yale University, 1993.

[15] D. Stewart. Combining Verilog Signals in Nets. Unpublished draft, available (as of 08-24-98) on the World-Wide Web at `http://www.cl.cam.uk/users/djs1002/verilog.project/papers/combining_signals.ps.gz`.

[16] D. Stewart. Modelling Verilog Port Connections. Unpublished draft, available (as of 08-24-98) on the World-Wide Web at `http://www.cl.cam.uk/users/djs1002/verilog.project/papers/ports.ps.gz`.

[17] J.P. Van Tassel. A Formalization of the VHDL Simulation Cycle. In L.J.M. Claesen and M.J.C. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications*, pages 359–374. Elsevier Science Publishers B.V. (North-Holland), 1993.