A Formal Model of Several Fundamental VHDL Concepts

David M. Goldschlag*
Naval Research Laboratory

Abstract

This paper presents a formal model of several fundamental concepts in VHDL including the semantics of individual concurrent statements, and groups of those statements, resolution functions, delta delays, and hierarchical component structuring. Based on this model, several extensions to VHDL are proposed, including nondeterministic assignments and unbounded asynchrony. Nondeterminism allows the specification of environments and of classes of devices. This model naturally captures the meaning of composition of VHDL programs.

1 Introduction

When defining a formal semantics for a programming language, it is important to identify key concepts in the language, develop a good formalization of those concepts, and define the rest of the language around this formalization. Such an approach helps unify the formalization effort, and provides insight into the key semantic underpinnings of the language. Furthermore, models of key concepts may suggests extensions to the programming language which are consistent with its current structure, yet increase its expressiveness and utility.

This paper presents a formal model of several fundamental VHDL[10, 5, 8] concepts. VHDL is a DOD standard hardware description language, which has been gaining wide acceptance in the hardware design community. It is notable for allowing text based descriptions at many levels, from abstract behavioral views of designs to gate level circuits which incorporate precise timing behavior. VHDL and the similar hardware description language Verilog are based on the notion of event driven simulation. VHDL is inherently concurrent, and allows the specification of hard real time properties. The formalization device used here is a translation to logic in the style of [6]. This approach seems very well suited to the semantics of VHDL.

A formal semantics for a programming language is essential for several reasons. If a language's semantics are ambiguous, there is little hope that compilers, simulators, and other programming support tools from different vendors will be completely compatible. Furthermore, without a precise definition, users of a language have no solid foundation upon which to analyze the meaning of their programs. Finally, if one would like to formally verify programs, a formal semantics is essential.

This paper is organized in the following way. Section 2 presents an informal introduction, by means of an example program, to the semantics of VHDL; this semantics is formalized in section 3. More of the language is formalized in section 4. Section 5 models VHDL's hierarchical approach to programming. Section 6 proposes several useful extensions to VHDL suggested by the formal semantics presented here, including nondeterministic assignments and unbounded asynchrony. Finally, section 7 offers some concluding remarks and comparison to related work.

2 An Informal Semantics

Consider the following fragment of a VHDL program:

```
x <= y + 1 after 10
```

(This program lacks the declarations for `x` and `y`, and other syntactic sugar required in a real VHDL program. The `<=` is VHDL's symbol for assignment.)

This program is composed of a single *concurrent signal assignment statement*. This sort of statement is fundamental to VHDL. In this example, the *signal* `x` (signals differ from variables) is assigned a value one greater than (the current value of) `y` after a delay of ten time units. The time units used are not relevant.

As mentioned earlier, VHDL is an event driven simulation language. This means that in each simulation cycle, concurrent statements are only executed if the signals that they depend on have changed since the last simulation cycle. In this case, if `y` has changed, the concurrent signal assignment statement computes `y + 1` and schedules signal `x` to get that value ten time units later.

How is this scheduling accomplished? A signal, unlike a simple variable, does not possess a simple value. Rather, a signal's value is obtained from its *driver* which is a set of `(time, value)` pairs called *events*; the value of a signal at time `t` is the *value* component

*Author's Address: Naval Research Laboratory, Code 5543, Building 16, Room 241, 4555 Overlook Avenue, S.W., Washington, D.C. 20375-5337, e-mail: goldschlag@itd.nrl.navy.mil.

of the **(time, value)** pair where **time** is the greatest *time* component not greater than **t**. An equivalent view is that a driver represents how a signal's value changes over time, and the value of the signal between two changes is the previous change.

At the end of the simulation cycle, the current time is increased to the time of the earliest next scheduled event among all drivers for all signals. Time, therefore, may increase by different amounts after each simulation cycle. The simulation cycle then repeats.

In this example, the statement implicitly waits for changes to the signal **y** (i.e., the statement waits for changes to any signal mentioned in the statement's right hand side).

The other executable statement in VHDL is the *concurrent process statement.* This construct is a sequential set of signal assignment statements with annotations identifying where the sequential execution suspends, and on which signals the statement subsequently waits. The concurrent signal assignment statement is really a single statement instance of the concurrent process statement, where the waiting annotation explicitly lists the signals on the assignment's right hand side.

3 A Formal Model

Before developing the formal model, it is instructive to make two observations:

- An important feature of signals is that an assignment to a signal never affects that signal's value during the current simulation cycle, so the evaluation of concurrent statements may occur in any order, or even simultaneously.
- Although the previous section stated that at the end of a simulation cycle, time is increased to the time of the earliest next scheduled event, such large increases are done in one step for the sake of efficiency only. If time were increased by any lesser amount, the next simulation cycle would essentially be a *skip* operation, since no signals would change.

A consequence of the second observation is that statements are really nondeterministic. This means that a statement may have multiple effects. For example, time increases (by an arbitrary amount) but may not skip the next event. Extending this insight further suggests that an individual statement only restricts the behavior of the signal being updated. All other signals may be modified in an arbitrary way. The behavior of other signals are themselves restricted by other statements in the program.

This approach suggests that signals not mentioned in a program are free to change in arbitrary ways. Although this is not how a VHDL simulator works, this view is perfectly consistent with the semantics of VHDL, since those (unmentioned) signals are not relevant to the program's behavior. If two programs are composed together (i.e., their concurrent process statements are merged) each component restricts the behaviors of its own signals. Signals from the environment should also be constrained by programs describing the behavior of the environment.

3.1 The Untimed Model

This approach may be more easily explained in the context of an (imaginary) untimed version of VHDL, which allows us to ignore both time and drivers. Consider the following program fragment, with two concurrent signal assignment statements:

```
y <= x
x <= y
```

As a result of each simulation cycle, the signals **y** and **x** swap values. The first concurrent signal assignment statement may be translated to the following logical formula:

```
y' = x
```

This formula means that the value of **y** in the new state (i.e., at the end of the current simulation cycle) will be the old value of **x**. The concurrent signal assignment to **x** may also be translated to:

```
x' = y
```

What formula captures the meaning of the program containing both signal assignment statements? It is the conjunction of the above two formulas:

```
y' = x
AND
x' = y
```

Each formula individually only restricts the behavior of the primed signal; together they restrict the behavior of both signals. In this formalization, conjunction is the appropriate composition operator, since the addition of more program statements serves to restrict the behaviors of the program. (The resulting program is more determined.)

The conjunction describes the behavior of a single simulation cycle for this program. To model the next simulation cycle, the unprimed variables are updated with the values of their primed counterparts, and the formula is applied again.

VHDL does not permit multiple concurrent signal assignment statements which assign to the same signal, unless one resolves the potential conflict by introducing resolution functions. Resolution functions are formalized in section 4.2.

3.2 The Timed Model

A similar construction for VHDL must permit both time and drivers. Consider the original program:

```
x <= y + 1 after 10
```

This formalization assumes the existence of a distinguished variable `time` which does not conflict with the names of any other state in the program. The translation is more complicated:

```
x' = IF has_changed(y, time) THEN
        update_inertial(
          x, value(y, time) + 1,
          time, time + 10)
      ELSE x
AND
time' > time
AND
time' ≤ next_event(y', time)
```

Each concurrent statement in the program is translated in this way; the semantics of the entire program is the conjunction of the translations of each component.

Although this translation is significantly more complicated than the untimed model, the form of any translated concurrent signal assignment statement will be similar. Each such formula will contain three conjuncts, each contributing part of the semantics:

- The first conjunct specifies how the driver for `x` changes: if the value of `y` has not just changed at time `time`, then the driver for `x` is unchanged as well. If however, the value of `y` has just changed, then the driver for `x` is updated with the new time and value pair. (The driver is updated according to the rules for *inertial* delays, the default in VHDL, which will not be defined here.)
- The second conjunct states that `time` must increase. But by how much?
- `time` may not become arbitrarily large. With respect to this particular statement, `time` must not skip the next scheduled event on the updated value of `y`. Notice that `y'` is used instead of `y`. This is because some other statement may schedule a new future event on `y` during this simulation cycle, before the existing next event on `y`.

  Notice that this conjunct works in concert with similar conjuncts from the translation of other statements. Taken together, `time` will not increase past the time of the next scheduled event among all the drivers of all the signals which are depended upon by statements in the program (as is required in section 2).

  In particular, one may ask whether this next event on `y` would indeed have been the next event on `y` once that future time is reached. The answer is that if it is not, it is because some earlier statement conspired to schedule an earlier event on `y`; so `time` would have been increased to not greater than the time of that earlier statement.

  Finally, it is important to note that although the two conjuncts about `time` guarantee that `time` will increase but will not increase too much, these two conjuncts do not imply that `time` will be set to the time of the next scheduled event. Rather, `time` may increase by some lesser amount. The next simulation cycle would then essentially be a *skip* operation, except that `time` will be increased further. Ultimately, if `time` has non-Zeno behavior,[1] `time` will increase to the time of the next scheduled event.

4 More VHDL

A good way to determine whether this formalization is appropriate for VHDL is to see whether it is easily extended to formalize more of the language. This section describes how to translate VHDL's *delta delays* and *resolution functions* within the logical formalism.

4.1 Delta Delays

In addition to the delay on signals described above, VHDL has a notion of a delta delay. An event which is scheduled with a delta delay will occur in the very next simulation cycle, before any previously scheduled event. Delta delays are often used to approximate the delays of gates within combinational circuits; sometimes, two VHDL designs are considered equivalent if the signals have the same values when the circuit becomes quiescent (at non-delta delay times). Delta delays are also used at higher levels of abstraction where the designer wants to specify only causal behavior.

Delta delays can be modeled here by making time into a pair, and using a lexicographic measure, as in SDVS[2]. That is, time would have two components: `(real, delta)`.[2] Delta delays increment the second component by one; real delays increase the first component and set the second component to zero. The lexicographic measure comparing two times `(t1, d1)` and `(t2, d2)` is:

```
(t1, d1) < (t2, d2) iff t1 < t2 OR
                        (t1 = t2 AND d1 < d2)
```

Delta delays allow the non-Zeno condition on time to be violated, even if both components are non-Zeno. A reasonable solution is to postulate the existence of some maximum bound on the size of the delta component and prove that it is consistent with some simulation of the program.

[1] `time` is required to have this property. Non-Zeno means that if a value increases forever, it will approach infinity. One easy way to satisfy this requirement is to make `time` an integer.

[2] `real` here does not imply that time is a real number.

4.2 Resolution Functions

The examples so far have considered signals which have only one driver (i.e., each signal occurs on the left hand side of only one signal assignment statement). Sometimes, a signal, like a *bus* or a *wired or*, might be driven by several inputs, but it is not natural to state this merger as a concurrent signal assignment statement in VHDL. In these cases a resolution function is used. A resolution function defines the value of the signal in terms of its drivers. In this formalization, a resolution function can be formalized as an invariant between driver values and the resolved signal. The invariant would be conjoined to the translation of the rest of the program.

5 Hierarchy

Hardware is often designed from components. These components may contain internal state. When a component occurs in several parts of the design, that internal state is not shared. This section presents a translation of VHDL components into logic, using the untimed model for the sake of simplicity.

Consider the following program fragment representing a buffer component containing two concurrent signal assignment statements:

```
internal <= input
output <= internal
```

This component links **input** and **output** signals by means of an internal wire named **internal**. In practice, the signals **input** and **output** will be mapped to external wires. The logical translation would be:

```
Lambda(input, output)(Exists internal ::
                          internal' = input
                          AND
                          output' = internal)
```

When this component is used, the **Lambda** expression is applied to the external wires, thereby instantiating the names **input** and **output**. Since the internal wire is existentially quantified, its name does not conflict with other wires. This component approach can be used hierarchically to reuse components at any level of nesting.

6 Extensions

This section proposes extensions to VHDL. Since this formalization allows nondeterminism, both timing delays and assignments can be made nondeterministic. For example, using **between** instead of the modifier **after** to specify delays implies that the new event is scheduled at some point in that interval. Consider a variant on the earlier example:

```
x <= y + 1 between (5, 10)
```

This statement may be formalized in the following way:

```
(Exists delay : 5 ≤ delay ≤ 10 :
        x' = IF has_changed(y, time) THEN
                update_inertial(
                  x, value(y, time) + 1,
                  time, time + delay)
                ELSE x)
AND
time' > time
AND
time' ≤ next_event(y', time)
```

Another extension allows unbounded delays:

```
x <= y + 1 later
```

This statement may be formalized in the following way:

```
(Exists delay : delay > 0 :
        x' = IF has_changed(y, time) THEN
                update_inertial(
                  x, value(y, time) + 1,
                  time, time + delay)
                ELSE x)
AND
time' > time
AND
time' ≤ next_event(y', time)
```

The **later** modifier specifies that the event will happen at some unspecified point in the future. **later** is especially interesting since it unifies the hard real time schedules that VHDL can already handle with completely asynchronous, yet fair[3], systems.

Another extension allows nondeterministic assignments. For example, VHDL cannot be used to generate a completely arbitrary sequence of positive numbers. The following statement (in the untimed model) would do so:

```
r in Nat
```

This would be translated as:

```
r' > 0
```

Other extensions may specify that signals are assigned from smaller sets.

The main difficulty with these extensions is that the resulting language is no longer simulatable. However, these types of extensions permit the specification of the timing properties of classes of devices (or the permitted tolerances in an implementation), and the specification of the expected environment of a system, and therefore significantly enhance the expressiveness of VHDL.

7 Conclusion

Several other researchers have been investigating the formalization of VHDL. Russinoff[9] has encoded an operational semantics of the language on the Boyer-Moore theorem prover[1]. Van Tassel[11] has developed a structured operational semantics for a subset which has been embedded on the HOL theorem prover[4]. Filippenko[2] is continuing to extend his formalization of VHDL within the SDVS system[7]. The approach taken here is novel because it translates programs to logical relations, and shows that conjunction, the restriction of the behavior of components, is a good model for VHDL.

This paper presents a formal model of several fundamental concepts in VHDL, including: resolution functions, delta delays, hierarchical component structuring, and the concurrent signal assignment statement. The collection of concurrent statements within a single VHDL program, and in fact, the composition of VHDL programs in general is easily modeled by the conjunction of the translations of the components.

The same techniques introduced here may be used to support a formalization of larger subsets of VHDL. Concurrent parts of a program, be they signal assignment or process statements, are combined by conjoining their translations.

This paper also presents several extensions to VHDL, taking advantage of nondeterminism. Nondeterminism allows the specification of the behavior of environments, both with respect to their processing of data, and their timing requirements. These extensions can also be used to specify classes of devices, or the tolerances within which devices must perform. Although such specifications are not simulatable, they are important and cannot be currently stated within VHDL.

The ability to specify environments is crucial for formal verification. When a VHDL program is tested, a set of test vectors is developed, each of which represent an instance of the behavior of the program's environment over time. For formal verification, this environment should itself be modeled by a program; the test vectors should be possible simulations of that program.

The modeling of VHDL's hard real time properties within this nondeterministic framework provides a simple means for unifying both bounded and unbounded delays. Although this does not imply that unbounded delays are really safety properties, being able to state them within the same (timed) framework is surprising.

This formalization provides a foundation for a formal semantics for VHDL, which could be used as a formal standard for the language and could support formal program verification. The process of formalizing even parts of the language provides useful insights about the language too.

References

[1] R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, Boston, 1988.

[2] I.V. Filippenko. Vhdl verification in the state delta verification system (sdvs). In P.A. Subrahmanyam, editor, *1991 International Workshop on Formal Methods in VLSI Design.* Springer-Verlag, 1991.

[3] Nissim Francez. *Fairness.* Springer-Verlag, New York, 1986.

[4] M. Gordon. Hol: A proof generating system for higher-order logic. Technical Report 103, University of Cambridge, Computer Laboratory, 1987.

[5] IEEE. *Draft Standard VHDL Langauage Reference Manual.* IEEE, New York, 1993.

[6] Leslie Lamport. The temporal logic of actions. Technical Report 79, DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, 1991.

[7] Leo Marcus. Sdvs 11 user's manual. Technical report, The Aerospace Corp., 1992.

[8] Douglas L. Perry. *VHDL, Second Edition.* McGraw-Hill, New York, 1994.

[9] David Russinoff. A formalization of a subset of vhdl. Technical report, Computational Logic, Inc., 1993.

[10] DAS Subcommittee. *IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1987.* IEEE, Inc., New York, 1987.

[11] John Peter Van Tassel. *Femto-VHDL: The Semantics of a Subset of VHDL and its Embedding in the HOL Proof Assistant.* PhD thesis, University of Cambridge, 1993.