

Resolution of Simulation Cycle Semantics In A Mixed Verilog-VHDL Simulation Environment

Victor Berman, Jay Lawrence

Cadence Design Systems, Inc.

1.0 Abstract

In past work the need for mixed language simulation environments to meet the needs of todays designers has been made clear [1], and several areas of potential conflict between the two languages have been illustrated including time-scale resolution, type mapping, and synchronization. This paper concentrates on the issue of resolving conflicts in the semantics of the simulation cycle between Verilog and VHDL to provide a well defined synchronization mechanism that ensures predictable event scheduling as simulation activity passes across the language interface. This work is based on research being done at Cadence to resolve semantic issues for a mixed language simulator currently under development. The paper shows how the native code approach to simulator implementation allows a feasible solution to the problem of mixed language semantic resolution.

2.0 Overview - Interoperability

The definition of VHDL/Verilog interoperability can be thought in terms of:

- Hierarchy - How are instances between languages specified, parameterized, connected, and bound?
- Simulation Cycle - What is the precise semantic of co-execution?
- Value Conversion - What state/strength conversions occur when values are passed between languages?

- Time Scale Resolution - how is the notion of time share between the two languages?
 - Name Space Resolution - How are names interpreted?
- While these are all important and have generally been discussed in [1], the issue of the definition of the shared simulation cycle is seen as key to providing a well behaved, predictable simulation tool which is useful to the circuit designer.

3.0 Simulation Cycle.

VHDL and Verilog have significantly different simulation cycles. The definition of how these simulation cycles interact is given in terms of the interleaving of concurrent execution and value scheduling semantics of both languages. The simulation cycle must guarantee that if only one language is present in the simulation, that the results of the simulation are the same as a single language simulator (ignoring language indeterminism). In addition, the cycle must be designed to eliminate overhead caused by extra synchronization steps between languages.

The following sections define an algorithm which can be used as a reference model for the behavior of VHDL/Verilog interaction. A given co-execution environment could use alternative algorithms if they are found to be more optimal as long as the resulting behavior is the same as the reference model.

The simulation cycle reference model is defined in terms a series of event lists: how they are created, how various event are added, how these event lists are updated, and how the evaluation of statements adds/removes to/from these lists.

In the remainder of the section, the VHDL definition a process has been extended to include Verilog always blocks, initial blocks, continuous assignments, UDPs, asynchronous tasks, and gate primitives. Each of these is potentially a concurrently executing behavioral element. The term “HDL process” will be used to refer to all this constructs. Also, the term HDL signal will be used to refer to any construct which can logically connect any two HDL processes. These include VHDL signals, and Verilog rags, nets, and named events.

3.1 Events

VHDL and Verilog simulators are both “event” driven. This means that HDL processes are evaluated in response to events. Classical events in Verilog can be changes in the value of nets or registers, or the triggering of a named event. In VHDL, events are the change in value of a signal or an implicit signal. Both these classes of events are defined here as *update* events. A special class of update events are those generated by Verilog non-blocking assignments, these are defined here as *M.B.A.-update* events. These M.B.A.-update events must be evaluated after all other events for a given time, although they may cause further zero-delay events to be scheduled. A second special class of update events are the *forcing -update* events, when a forcing update event is evaluated for a given HDL signal, then all other update events for the given HDL signal are removed from all future cycle and time lists. These forcing-update events can only be removed by another forcing-update event on the same HDL signal, or an explicit deassign or release evaluation event.

It is also convenient for the purpose of defining the simulation cycle to treat the execution of HDL process as events. This class of events is defined here as *evaluation* events.

Finally, the execution of wire-tie, resolution, or type-conversion functions is a final kind of event. These events are defined here as *propagation* events.

3.2 Simulation Reference Model

We will use the term simulation time to represent the amount of actual time it would take for the circuit being simulated. In the reference model, the variable T represents the current simulation time. Because HDL models commonly contain zero delay statements, the simulation reference model must be further subdivided into simulation cycles within a simulation time. The current simulation cycle count within a given time is represented by the variable C.

In the simulator reference model, each possible simulation time contains each of the following event lists.

- *Init-Time-List* - consists of events which are executed first in any given time.
- *Time-List* - consists of events which must be executed during the simulation time.
- *Init-End-of-Time-List* - consists of update events which must be executed when the kernel believes in may have reached the end of a time. Note that execution of events on this list, may cause additional cycles at this time to occur.
- *End-of-Time-List* - consists of events which have been scheduled to execute after all other events for the current time. Note that execution of events on this list, may cause additional cycles at this time to occur.
- *Monitor-Time-List* - consists of update and evaluation events which are executed as the final set of events before simulation time advances. It is an error if execution of events on this list cause additional cycle at this time to occur.

In addition to the above lists for each simulation time, the simulator reference model maintains four variables which are set for each simulation cycle.

- *Init-Cycle-List* - consists of events which must be executed first in the current cycle. This is initially set to the *Init-Time-List* for any given time T.
- *Cycle-List* - consists of events which must be executed at the current cycle. This is initialized to the *Time-List* for any given time T.
- *Init-Next-Cycle-List* - consists of events which must be executed to initialize the next simulation cycle. This will only be added to by executing events on the *Cycle* list.
- *Next-Cycle-List* - consists of events which must be executed in the next cycle at this time. This will only be added to by executing events on the *Cycle* list.

The algorithm for the simulation reference model is as follows:

```

T = 0;
Add an evaluation event for all HDL processes to
the Time-List for T = 0;

while (Init-Time-List[T] or Time-List[T]) { /*
Execute a simulation time */

    Init-Cycle-List = Init-Time-List[T]
    Cycle-List = Time-List[T]
    C = 0;

    while (Init-Cycle-List or Cycle-List) {

        /* Execute a simulation cycle */
        Init-Next-Cycle-List = NULL;
        Next-Cycle-List = NULL;

        C++;
        execute events on Init-Cycle-List
        execute events on Cycle-List

        if (Init-Next-Cycle-List) {
            Init-Cycle-List = Init-Next-Cycle-List;
        }
        if (Next-Cycle-List) {
            Cycle-List = Next-Cycle-List;
        }

        /* Check if anything was found */
        if ( (Init-Cycle-List == NULL) and
            (Cycle-List == NULL)) {

            /* Try to end the time */
            if (Init-End-Of_Time[T]) {
                Init-Cycle-List =
                    Init-End-Of-Time-List[T];
            }
            else if (End-Of-Time[T]) {
                Cycle-List = End-Of-Time[T];
            }
            else if (Monitor-Time-List[T]) {
                Cycle-List =
                    Monitor-Time-List[T];
            }
        }
    }
}

```

```

} /* end if cycle lists are NULL */

} /* end while more in cycle lists */

/* Cycle is over, update time */
increment T to the next time with events

} /* end while there are events in a future time*/

```

3.3 Mapping HDL Constructs onto Event Lists

Simulation proceeds by HDL processes generating events, those being placed on one of the event queues, and then the simulation cycle progressing until those events are executed. This section outlines the events created by execution of various HDL statements, the event lists on which they are placed.

3.3.1 Continuous Assignment

A continuous assignment statement is an HDL process which is sensitive to the HDL signals on the right-hand side of the assignment. The evaluation of an update event on any of the sensitive HDL signals causes the continuous assignment evaluation event to be created. This evaluation event is placed on the Cycle-List. When this event is evaluated, an update event is generated for zero or more HDL signals on the left-hand side of the assignment. If there is no delay in the continuous assignment, this update event is placed on the Cycle-List. If a delay of #0 is present, the update event is placed on the Next-Cycle List. If a delay of greater than 0 is present the update event is placed on the Time-List for the appropriate time.

3.3.2 Blocking Assignment

Blocking assignments are found in behavioral blocks. If during the execution of an evaluation event, a blocking assignment is executed, an update event is created for one or more HDL signals on the left-hand side of the assignment. If the blocking assignment is an assign or force command, then the update event created is a forcing-update event. The value of the update event is not determined until the update event is evaluated, at that time the current values of the right-hand side primaries are used to determine the value. The currently executing evaluation event is set to be sensitive to the evaluation of the just created update event.

If there is no intra-assignment delay on the assignment, the update event is placed on the Cycle-List. If there is a zero-delay intra-assignment delay, the update event is placed on the Next-Cycle list, and if there is a delay greater than zero, the update event is placed on the Time-List at the appropriate time.

In the case where no delay is specified, the kernel may immediately evaluate the just created update event, thereby re-enabling the evaluation event continuing sequential evaluation, or may interleave evaluation of other events on the current Cycle-List.

3.3.3 Non-Blocking Assignment

Non-Blocking assignments are found in behavioral blocks. If during the execution of an evaluation event, a non-blocking assignment is executed, an update event is created for one or more HDL signals on the left-hand side of the assignment. The value of the update event is determined by current values of the right-hand side primaries. If there is no intra-assignment delay or a delay of zero on the assignment, the update event is placed on the Init-End-Of-Time-List for the current time. If there is a delay greater than zero, the update event is placed on the Init-End-Of-Time-List at the appropriate time.

The simulator may immediately continue the current evaluation event or may interleave evaluation of other events on the current Cycle-List.

3.3.4 Gate Primitives/UDPs

Gate primitives execution behavior are defined in terms of a continuous assignment statement which connects the inputs of the gate or UDP to the output of the gate or UDP. The functionality of the implicit continuous assignment is the functionality of the gate.

3.3.5 Named-Events

The triggering of a named-event, causes an update-event to be inserted on Cycle-List. When this event is evaluated, any evaluation events which are sensitive to the update event are placed on the Cycle-List.

3.3.6 VHDL Processes

VHDL process statements are executed in response to the evaluation of an update event for any HDL signal on which they have sensitivity. When an update event occurs on any signal to which the process is sensitive, an evaluation event for the process is placed on the Cycle-List. When this event

is evaluated, a propagate event is generated for zero or more HDL signals which are assigned to using signal assignment statements. If there is an implicit or explicit zero delay, the propagate event is placed on the Init-Next-Cycle-List. If a delay of greater than 0 is present the propagate event is placed on the Init-Time-List for the appropriate time. If the concurrent signal assignment is contains a multiple output waveform, then each waveform element is treated as an individual propagate event. Propagate events scheduled in this way may cause other propagate events to be removed from the Init-Next-Cycle or some future Init-Time list, if they propagate a value to the same signal and either transport or inertial de-scheduling need take place. If the process suspends due to the execution of a wait for <time> statement, then an evaluation event for the process is placed on the appropriate Time-List.

If a process is a postponed process, then the evaluation event is placed on the Monitor-Time list for current time instead of the Cycle-List.

3.3.7 Monitor and Strobe Statements

When a monitor or strobe statement executes, an evaluation event is placed on the Monitor-Time list for the current cycle. The evaluation of the monitor or strobe statement causes the HDL signals which are being monitored to be evaluated and their current values displayed. The evaluation event can also the evaluation event to be re-scheduled on a future Monitor-Time-List.

3.3.8 Propagation Event

Propagation events are always executed from the Init-Cycle list. These events cause the values of drivers for VHDL signals to be propagated to the actual signal. Many kinds of evaluation events may be sensitive to the evaluation of propagate events. When a propagate event executes, if a resolution function or type-conversion function exists for the signal, evaluation events for these behaviors are placed on the Init-Cycle list. These evaluations may in turn cause further propagate events to be scheduled, when all propagate events have executed, an update event is added to the Init-Cycle list for the signal. These update events when executed may cause evaluation events for any HDL processes which are sensitive to the signal to be added to the Cycle-List.

4.0 Implementation Considerations

In the previous section the definition of the combined simulation cycle semantic of the two HDLs is presented in terms of a list/queue reference model. This model provides an active protocol which may be used effectively in mapping language constructs to simulation actions. This reference model goes beyond the limited interaction model generally supported in traditional co-simulation paradigms which make use of some variant on back-plane technology. The advantage of this model is that it provides a framework for interaction semantics for the complete set of both language constructs. The difficulty with this model is that it imposes unique challenges for the implementor to ensure correct and efficient simulation behavior.

The traditional back-plane or even inter-locking kernel implementations will have difficulty providing full language coverage without a significant penalty in speed and memory usage due to the underlying limitations of a multi-kernel system. Such kernels have implicit notions of the simulation cycle semantic of the language they were designed to implement, and any extension is painfully costly, thus leading to the multi-kernel implementation. This solution has several drawbacks the most obvious being the necessity of maintaining and executing them. This both increases the cost and also reduces the opportunity for improving optimization.

The approach suggested here eliminates the need for a traditional kernel and instead substitutes an interleaved code stream of native machine instructions which directly implement the required language semantics.

4.1 Language Processing

In order to create highly optimized native code streams for the simulation of mixed language designs, the front-end language processors or parsers must be designed to produce compatible intermediate representations of the two languages. A common elaborator which is cognizant of the binding rules for both languages is used to process the

intermediate representations from both languages and produce the correct hierarchical representation of the design to be simulated. The figure below illustrates the architecture:

.Note that in this architecture two separate code generators are used to process the elaborated design, each working on the data in the design which was input in its particular language. However, these streams are then merged so that the final phases of machine dependent optimizations may be performed globally on the now language independent stream. This decomposition into language dependent/independent processing is just as critical for the success of the design as is the decomposition into machine dependent/independent processing. These decompositions are the key element in managing the complexity of the processing as well as in ensuring that the highest appropriate levels of optimizations can be realized.

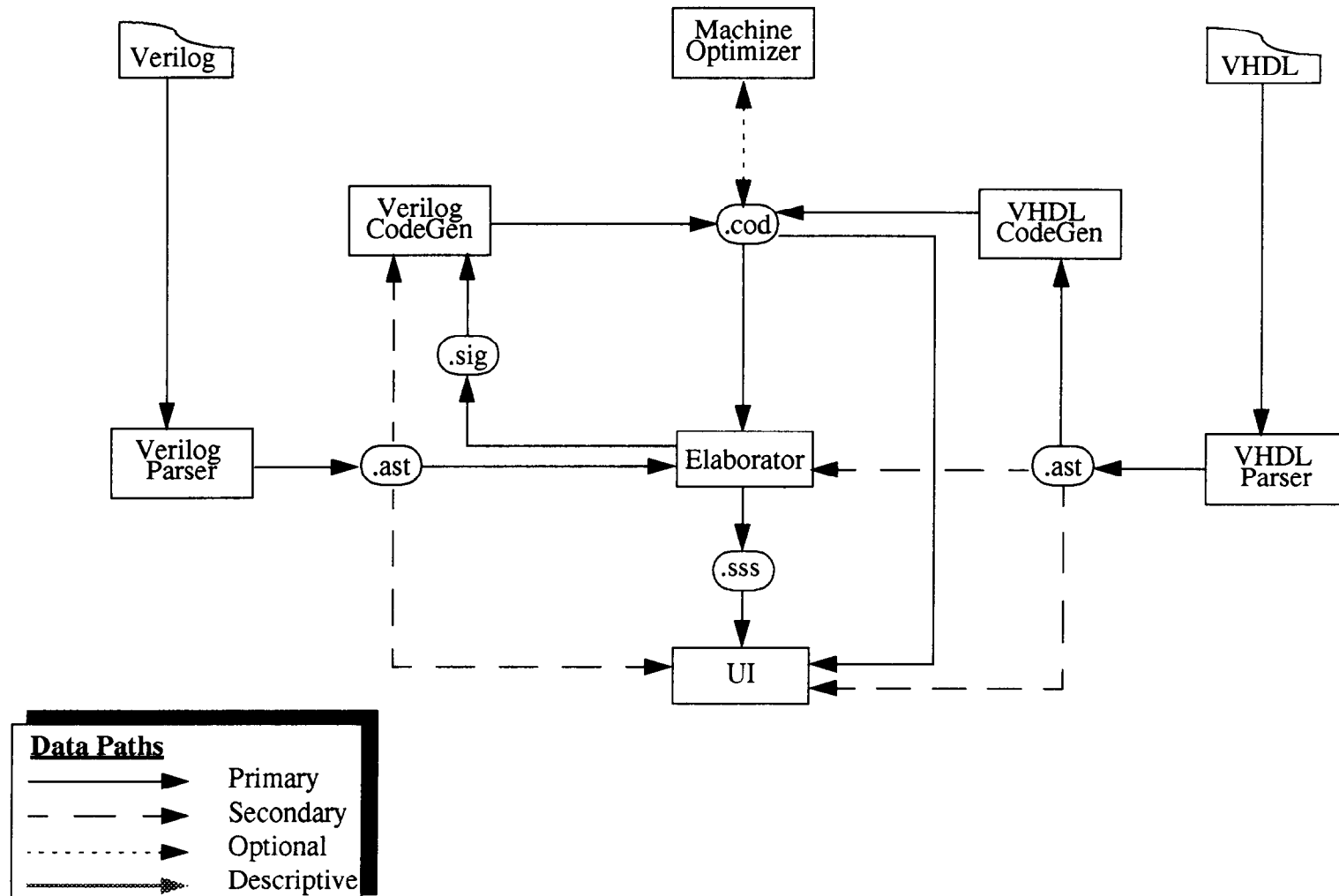
4.2 Elaboration

Notice that the architecture diagram shows an added path between the elaborator and the Verilog Code Generator. This asymmetry is due to the fact that elaboration data is not distributed in the same way in the two languages. VHDL makes a clear distinction between interface and implementation through the entity/architecture pair. This distinction is less formal in Verilog so the process of code generation must make use of both pre- and post-elaboration data in order to achieve separate compilation.

The elaborator must also be aware of the different binding possibilities when the two languages are combined. Quite a few varieties are possible depending on the parent/child relationships between the VHDL and Verilog instances. The complete syntax and specification of these elaboration hierarchies is out of the scope of this paper and will be treated in a later paper.

Multi-Language Processing Environment

11.8



5.0 Summary

This paper has developed a list/queue reference model for defining the semantics of mixed VHDL Verilog Simulation. It has further briefly discussed some of the issues in implementing such a system and outlined a feasible solution based on research which is ongoing at Cadence Design Systems. We look forward to presenting the results of this work and in sharing our findings with the IEEE 1076.5 Working Group which is developing standards for VHDL-Verilog interoperability.

6.0 Future Plans

There are a number of topics related to this paper which need to be explored and documented before the bi-lingual semantics are completely specified. Several have already been mentioned:

- Hierarchy
- Value Conversion
- Name Space Resolution
- Time Scale Resolution

While some work has been done in specifying these we plan to complete this work and report on it future papers. One area, that of hierarchy is of great interest since its correct solution may require changes to one or both languages before it is made fully useful and well specified.

7.0 References

- [1] Berman, Victor, "Standard VHDL Verilog Interoperability", Proceedings of the International Verilog Conference, 1994, San Jose, CA.
- [2] "IEEE Standard VHDL Language Reference Manual IEEE Std 1076-1987", 1988, The Institute of Electrical and Electronics Engineers, Inc. New York, NY.
- [3] "IEEE Draft Standard VHDL Language Reference Manual IEEE Std 1076-1992B", 1993, The Institute of Electrical and Electronics Engineers, Inc. New York, NY.
- [4] "Verilog Hardware Description Language Reference Manual, Version 2.0", March 1993, Open Verilog International, San Jose, CA.