A Formal Executable Semantics of Verilog

Patrick Meredith Michael Katelman
Grigore Roşu José Meseguer

University of Illinois at Urbana-Champaign

MEMOCODE 2010

What is the semantics of Verilog?

- IEEE Std. 1364-2005, VCS, NCSim, ModelSim, etc.
  - (comprehensive, but not formal)
- [Gordon95],[Pace98],[Sasaki99],[Zhu06]
  - (formal, but not comprehensive)
- we utilize advances in formal semantic frameworks to address scalability problems of previous formalization attempts.
  - our semantics is also executable.

IEEE Std. 1364-2005

Verilog designs and tools are ultimately governed by the official semantics of Verilog, IEEE Std. 1364-2005.

- The standard is written in non-mathematical English prose.
- It is comprehensive and generally clear, but not formal.
- Large, about 600 pages, reflecting the scale of the language.

Augmenting the Standard with a Formal Semantics

A formal semantics provides a uniform and precise model for reasoning about complex aspects of Verilog behavior.

- applicable for EDA tool developers and designers.

EDA Tool Developers

Implementing Verilog-based tools typically requires a deep understanding of Verilog semantics.

- especially formal tools providing strong behavior guarantees.
- EDA advances are crucial to continued progress in semiconductor industry.

Designers

Verilog is highly concurrent and designs can exhibit complex behaviors. Designers benefit from a clear mental model to understand their own designs and to use tools.

- a uniform model aids communication between engineers.
- especially useful for new users to avoid common mistakes.
- determine if tools are yielding correct results.

Propagation Loop

```
1   module propagation_loop;
2     reg [15:0] x;
3     initial
4     begin
5       x  = 1;
6       x  = 0;
7       x <= 2;
8       #10 $display("x = %d", x);
9      $finish;
10    end
11    always @(x[0])
12    begin
13      x = x + 1;
14    end
15  endmodule
```

Result

- VCS produces the answer **x = 3**
- Icarus Verilog (v.092) infinite loops
- Executing our semantics says both **x = 2** and **x = 3** are correct answers.

Net Assignment Nondeterminism

```
1   module net_assignment;
2    wire x;
3    reg y;
4    assign x = y;
5    initial
6    begin
7     y = 0; y = 1;
8     y = 0; y = 1;
9   end
10  always@(posedge x)
11    $display("posedge x");
12  endmodule
```

Result

- VCS displays the message once
- Icarus Verilog never displays the message
- Executing our semantics shows it is legal to print the message 1 or 2 times.

Non-Blocking Assignment

```
1   module nb_assignment;
2     reg [15:0] x,y;
3     initial
4     begin
5       y = 0;
6       x <= 0; x <= 1;
7       x <= 2; x <= 3;
8       #10
9   $display("y = %d", y);
10   end
11   always @(x[0])
12   begin
13     y <= y + 1;
14   end
15  endmodule
```

Result

- VCS **y** has the value **1**
- Icarus Verilog **y** has the value **2**
- Executing our semantics says that the value of **y** must be **1**.

IEEE Std. 1364

The language of the IEEE Std. uses a particular terminology.

- events, event scheduling, event execution.
- processes, process execution, suspension, and sensitivity.

Toward a Formal Verilog Semantics

The goal of a formal semantics for Verilog is to give this terminology a precise mathematical meaning.

- what is an **event**? what is a **process**?
- how are individual events and processes **executed**?
- how are events and processes **scheduled** for execution?
- how is **non-determinism** exhibited?

Events

For example, many different kinds of events are discussed; a formal semantics must say what the precise structure of each one is:

- active events - essentially the currently running processes
- inactive events - **0** delayed processes
- non-blocking assign update events - waiting non-blocking updates
- monitor events - events created by the **$monitor** function
- future events - events that will happen during a cycle at a future simulator time
- listening events - not defined in the standard, these are processes that are waiting for an input change

Semantic Frameworks

There are many frameworks for doing formal semantics, and one must make some choice.

- SOS and MSOS
- Context Reduction
- CHAM
- Rewriting Logic Semantics and K

Rewriting Logic Semantics

We use the framework of Rewriting Logic Semantics.

- We feel it has an intuitive definitional style
- Highly suited for semantics with concurrency
- There exist efficient tools for executing rewriting systems.
  - Secure feeling semantics are correct via testing
  - Gives us a tool useful for evaluating semantic questions

Brief Introduction to RLS

- Equations of the form $l = r$
- Rules of the form $l \longrightarrow r$
- Evaluation occurs by finding a rule or equation such that its l matches all or part of the term
- Equations can be thought of as deterministic instantaneous changes
- Rules are actual (possibly non-deterministic) state transitions
  - State update
  - Scheduling

Brief Introduction to RLS

- The *l* and *r* of an equation or rule contain variables which are *instantiated*
- We use the operator *k* to denote the computation of a process

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

Configuration

- To understand a RLS definition we must first understand its *configuration*
- The configuration is the term that represents the entire simulation state
- All evaluations are manifest as changes to the configuration

Configuration

A Small Module

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

Configuration

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

Always and Initial Procedural Blocks

- Initial blocks simply schedule their statement to run

$$\mathrm{top}(\textbf{initial}\ S) = \mathrm{stmt}(S)$$

- Always is essentially an infinite loop
- Must mention k operator to avoid infinite expansion

$$\mathrm{k}(\mathrm{top}(\textbf{always}\ S)) = \mathrm{k}(\mathrm{stmt}(S) \curvearrowright \mathrm{top}(\textbf{always}\ S))$$

Blocking Assignments

- Must store rest of computation and unschedule until after update occurs
- The actual update is a rule, as we will see

$$\begin{aligned} & \text{activeProcesses}(\text{k}(BV \curvearrowright \text{blockingAssign}(X) \curvearrowright K)\ PS) \\ & \text{updateEvents}(ES) \\ = \ & \text{activeProcesses}(PS) \\ & \text{updateEvents}(\text{updateEventList}(\text{updateEvent}(X, BV), K)\ ES) \end{aligned}$$

Non-Blocking Assignments

- We use a list because the standard mandates that non-blocking assignments within one procedural block must complete in order
- We use a rule so that non-blocking assignments between different procedural blocks will non-deterministically interleave

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

Continuous Assignment

Continuous assignments are very curious, and IEEE does not provide much guidance. E.g.,

```
assign x = y;
```

is not just

```
always @(y)
  x = y;
```

What happens if **y** changes first to **1** and then **2**?

Continuous Assignment

- Important issue here is that there can only be one outstanding update
- The standard is actually not clear on this point

$$
\begin{aligned}
&\text{activeProcesses}(\text{continuousk}(X, BV_1)\ PS) \\
&\text{updateEvents}(\text{continuousUpdateEvent}(X, BV_2)\ ES) \\
=\ &\text{activeProcesses}(PS) \\
&\text{updateEvents}(\text{continuousUpdateEvent}(X, BV_1)\ ES)
\end{aligned}
$$

$$
\begin{aligned}
&\text{activeProcesses}(\text{continuousk}(X, BV)\ PS) \\
&\text{updateEvents}(ES) \\
=\ &\text{activeProcesses}(PS) \\
&\text{updateEvents}(\text{continuousUpdateEvent}(X, BV)\ ES) \\
&\quad \textit{otherwise}
\end{aligned}
$$

Lookup and Updates

- Must be a rule to facilitate non-determinism

$$\begin{aligned} &\text{activeProcesses}(\text{k}(\text{exp}(N, X) \curvearrowright K)\ PS)\ \text{env}(X \leftarrow BV, Env) \\ \rightarrow\ &\text{activeProcesses}(\text{k}(BV \curvearrowright K)\ PS)\ \text{env}(X \leftarrow BV, Env). \end{aligned}$$

- *sense* operator wakes up listening processes and monitors

$$\begin{aligned} &\text{updateEvents}(\text{updateEventList}(\text{updateEvent}(X, BV_1); EL, K) ES_1) \\ &\text{monitorEvents}(ES_2) \\ &\text{env}(X \leftarrow BV_2, Env) \\ &\text{listeningEvents}(ES_2) \\ \rightarrow\ &\text{updateEvents}(\text{updateEventList}(EL, K) ES_1) \\ &\text{env}(X \leftarrow BV_1, Env) \\ &\text{listeningEvents}(\text{sense}(X, BV_2, BV_1, ES_2)) \\ &\text{monitorEvents}(\text{sense}(X, BV_2, BV_1, ES_3)) \end{aligned}$$

Scheduling Inactive Events

- Inactive events are schedule only where there are no active processes or update events.

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

Scheduling Non-Blocking Assignment Updates

- Non-Blocking assignment updates are scheduled only when there are no active processes and no inactive events

activeProcesses(empty)
updateEvents(empty)
inactiveEvents(empty)
nonBlockingAssignUpdate(NEL)
$\rightarrow$ activeProcesses(empty))
updateEvents(updateEventList(NEL, empty))
inactiveEvents(empty)
nonBlockingAssignUpdateEvents(empty)

Conclusions

- Most complete formal definition of Verilog we are aware of.
- Also, the only executable formal definition
  - Increases confidence in definition
  - Useful tool for determining proper output
- Useful as a starting point for further discussion on Verilog semantics