A Formal Executable Semantics of Verilog

Patrick Meredith    Michael Katelman
Grigore Roșu       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 \texttt{y} has the value 1
- Icarus Verilog \texttt{y} has the value 2
- Executing our semantics says that the value of \texttt{y} must be 1.
The language of the IEEE Std. uses a particular terminology.

- events, event scheduling, event execution.
- processes, process execution, suspension, and sensitivity.
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?
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
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 \( I = r \)
- Rules of the form \( I \rightarrow r \)
- Evaluation occurs by finding a rule or equation such that its \( I \) 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

$$k(\text{stmt}(\text{begin } S SL \text{ end} \leadsto K))$$
$$= k(\text{stmt}(S) \leadsto \text{stmt}(\text{begin } SL \text{ end}) \leadsto K)$$
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.
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

```plaintext
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

\[
\text{top(} \text{initial } S \text{)} = \text{stmt}(S)
\]

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

\[
k(\text{top(} \text{always } S \text{)}) = k(\text{stmt}(S) \leadsto \text{top(} \text{always } S \text{)})
\]
Blocking Assignments

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

\[
\begin{align*}
\text{activeProcesses}(k(BV \rightarrow \text{blockingAssign}(X) \rightarrow K) \ PS) \\
\text{updateEvents}(ES) \\
= \text{activeProcesses}(PS) \\
\text{updateEvents}(\text{updateEventList}(\text{updateEvent}(X, BV), K) \ ES)
\end{align*}
\]
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.

\[
\text{activeProcesses}(k(BV \rightarrow \text{nonBlockingAssign}(X) \rightarrow K) \text{ PS}) \\
\text{nonBlockingAssignUpdateEvents}(EL) \\
\rightarrow \text{activeProcesses}(k(K) \text{ PS}) \\
\text{nonBlockingAssignUpdateEvents}(EL; \text{updateEvent}(X, BV))
\]
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 \( 2 \)?
Important issue here is that there can only be one outstanding update

The standard is actually not clear on this point

\[
\begin{align*}
\text{activeProcesses(continuous}(X, BV_1) PS) & \\
\text{updateEvents(continuousUpdateEvent}(X, BV_2) ES) & = \text{activeProcesses}(PS) \\
& \text{updateEvents(continuousUpdateEvent}(X, BV_1) ES) \\
\text{activeProcesses(continuous}(X, BV) PS) & \\
\text{updateEvents}(ES) & = \text{activeProcesses}(PS) \\
& \text{updateEvents(continuousUpdateEvent}(X, BV) ES) \\
\text{otherwise} &
\end{align*}
\]
Lookup and Updates

- Must be a rule to facilitate non-determinism

\[
\text{activeProcesses}(k(\exp(N, X) \bowtie K) \ PS) \ env(X \leftarrow BV, Env) \\
\rightarrow \text{activeProcesses}(k(BV \bowtie K) \ PS) \ env(X \leftarrow BV, Env).
\]

- \textit{sense} operator wakes up listening processes and monitors

\[
\text{updateEvents}(\text{updateEventList}(\text{updateEvent}(X, BV_1); EL, K) ES_1) \\
\text{monitorEvents}(ES_2) \\
env(X \leftarrow BV_2, Env) \\
\text{listeningEvents}(ES_2) \\
\rightarrow \text{updateEvents}(\text{updateEventList}(EL, K) ES_1) \\
env(X \leftarrow BV_1, Env) \\
\text{listeningEvents}(\text{sense}(X, BV_2, BV_1, ES_2)) \\
\text{monitorEvents}(\text{sense}(X, BV_2, BV_1, ES_3))
\]
Inactive events are schedule only where there are no active processes or update events.

\[
\begin{align*}
\text{activeProcesses} & (\text{empty}) \\
\text{updateEvents} & (\text{empty}) \\
\text{inactiveEvents} & (NES) \\
\rightarrow \text{activeProcesses} & (\text{activate}(NES)) \\
\text{updateEvents} & (\text{empty}) \\
\text{inactiveEvents} & (\text{empty})
\end{align*}
\]
Non-Blocking assignment updates are scheduled only when there are no active processes and no inactive events

\[
\text{activeProcesses}(\text{empty}) \\
\text{updateEvents}(\text{empty}) \\
\text{inactiveEvents}(\text{empty}) \\
\text{nonBlockingAssignUpdate}(NEL) \\
\rightarrow \text{activeProcesses}(\text{empty})) \\
\text{updateEvents}(\text{updateEventList}(NEL, \text{empty})) \\
\text{inactiveEvents}(\text{empty}) \\
\text{nonBlockingAssignUpdateEvents}(\text{empty})
\]
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