Directed Testing as Meta-Reasoning about Simulation

Michael Katelman
(joint work with José Meseguer)

University of Illinois at Urbana-Champaign

September 12, 2009
Simulation dominates industrial verification methodology.

- constrained random testbench, large compute cluster
- \( \approx 95\% \) of bugs found through simulation (ITRS 2007/2008)

Enormous engineering effort is required to build, constantly adjust, and maintain the testbenches.

- “...sources report that in current development projects verification engineers outnumber designers, with this ratio reaching two to one for the most complex designs.”
  
  (ITRS 2007/2008)

- improving simulation improves the overall verification effort
Various types of testbench:

- **directed testbench** = writing a test by hand (not automated)
- random testbench = constraints used to generate a test

Creating a **directed testbench** is extremely tedious and difficult

However, directed testing is a reality of verification.
always @(posedge clk)
begin
  case (position)
    0 : position <= i ? 7 : 1;
    1 : position <= i ? 2 : 7;
    2 : position <= i ? 7 : 3;
    3 : position <= i ? 4 : 7;
    4 : position <= i ? 7 : 5;
    5 : position <= i ? 6 : 7;
    6 : position <= 6; // ‘‘out’’
    7 : position <= 7; // ‘‘dead-end’’
  end
end

directed testbench:

#0 clk = 0;
#0 i = 1;

#5 clk = 1;
stuck! Pointless to continue simulation from here . . .

#5 clk = 0;
#0 i = 0;
#5 clk = 1;
#5 clk = 0;
#0 i = 1;
#5 clk = 1;
...

Michael Katelman
Directed Testing as Meta-Reasoning about Simulation
Maze

Example

always @(posedge clk)
begin
    case (position)
        0 : position <= i ? 7 : 1;
        1 : position <= i ? 2 : 7;
        2 : position <= i ? 7 : 3;
        3 : position <= i ? 4 : 7;
        4 : position <= i ? 7 : 5;
        5 : position <= i ? 6 : 7;
        6 : position <= 6; // ‘out’
        7 : position <= 7; // ‘dead-end’
    end
end

directed testbench:

#0 clk = 0;
#0 i   = 1;

#5 clk = 1;

stuck! Pointless to continue simulation from here . . .

#5 clk = 0;
#0 i   = 0;
#5 clk = 1;
#5 clk = 0;
#0 i   = 1;
#5 clk = 1;

...
Example

```verilog
case (position)
  0 : position <= i ? 7 : 1;
  1 : position <= i ? 2 : 7;
  2 : position <= i ? 7 : 3;
  3 : position <= i ? 4 : 7;
  4 : position <= i ? 7 : 5;
  5 : position <= i ? 6 : 7;
  6 : position <= 6; // "out"
  7 : position <= 7; // "dead-end"
end
```

directed testbench:

```verilog
#0 clk = 0;
#0 i = 1;
#5 clk = 1;

stuck! Pointless to continue simulation from here . . .

#5 clk = 0;
#0 i = 0;
#5 clk = 1;
#5 clk = 0;
#0 i = 1;
#5 clk = 1;
...
```
Everyone knows the programmatic solution; *backtracking!*

```haskell
solveMaze p is vs done = do
    when (p == 6) (done (is,p:vs))
    if p `elem` vs
        then return (tail is,vs)
        else do
            (_,v0s) <- solveMaze (f 0 p) (0:is) (p:vs) done
            id solveMaze (f 1 p) (1:is) (v0s) done
```

...*this is not possible with a Verilog testbench!*
Everyone knows the programmatic solution; *backtracking*!

```plaintext
solveMaze p is vs done = do
  when (p == 6) (done (is,p:vs))
  if p 'elem' vs
    then return (tail is,vs)
  else do
    (_,v0s) <- solveMaze (f 0 p) (0:is) (p:vs) done
    id solveMaze (f 1 p) (1:is) (v0s) done
```

...this is not possible with a Verilog testbench!
Shortcomings of Directed Testing Languages

- Cannot backtrack. Time moves in one direction only.
- No coordination across multiple simulation runs.
- Cannot conditionalize current input choices based on events that occur in the future.
- No symbolic simulation (or associated automated solving capabilities).

Main Question!

Can we define a language that addresses the shortcomings above?
Shortcomings of Directed Testing Languages

- Cannot backtrack. Time moves in one direction only.
- No coordination across multiple simulation runs.
- Cannot conditionalize current input choices based on events that occur in the future.
- No symbolic simulation (or associated automated solving capabilities).

Main Question!
Can we define a language that addresses the shortcomings above?
Formally define a new language for directed testing.
- Idea is to have *symbolic simulation* a *first-class object* that can be manipulated programatically.

Demonstrate how the language addresses the above issues.

Briefly describe some specific applications to microprocessors and other types of hardware.

Describe a tool that we are building for Verilog designs.
Formally define a new language for directed testing.
- Idea is to have *symbolic simulation* a *first-class object* that can be manipulated programmatically.

Demonstrate how the language addresses the above issues.

Briefly describe some specific applications to microprocessors and other types of hardware.

Describe a tool that we are building for Verilog designs.
Formally define a new language for directed testing.

- Idea is to have *symbolic simulation* a *first-class object* that can be manipulated programmatically.

Demonstrate how the language addresses the above issues.

Briefly describe some specific applications to microprocessors and other types of hardware.

Describe a tool that we are building for Verilog designs.
Idea is to have *symbolic simulation* a *first-class object* that can be manipulated programmatically.

Demonstrate how the language addresses the above issues.

Briefly describe some specific applications to microprocessors and other types of hardware.

Describe a tool that we are building for Verilog designs.
1. Formally define a new language for directed testing.
   - Idea is to have *symbolic simulation* a *first-class object* that can be manipulated programmatically.

2. Demonstrate how the language addresses the above issues.

3. Briefly describe some specific applications to microprocessors and other types of hardware.

4. Describe a tool that we are building for Verilog designs.
Formally, our language is designed so that testing becomes an exercise in *meta-reasoning* about *symbolic simulation*.

<table>
<thead>
<tr>
<th>Testing Language (Rewriting Logic)</th>
<th>ML &amp; LCF</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{R}_{RTL}$</td>
<td>Any set of axioms in LCF.</td>
</tr>
<tr>
<td>(axiomatization of the RTL semantics)</td>
<td></td>
</tr>
<tr>
<td>$\mathcal{R}_{IR}$</td>
<td>The inference rules of LCF.</td>
</tr>
<tr>
<td>(axioms about $\hat{\mathcal{R}}_{RTL}$)</td>
<td></td>
</tr>
<tr>
<td>$\mathcal{R}_{STRAT}$</td>
<td>An ML program.</td>
</tr>
<tr>
<td>(A testing strategy. $\mathcal{R}<em>{IR} \subseteq \mathcal{R}</em>{STRAT}$)</td>
<td></td>
</tr>
</tbody>
</table>

- $\mathcal{R}_{IR}$ and Rewriting Logic constitute “the language” ($\approx$ ML).
\( \mathcal{R}_{RTL} \) defines a relation (via the inference rules of rewriting logic) between program configurations:

\[
\mathcal{R}_{RTL} \vdash \langle p_1, \sigma_1 \rangle \longrightarrow \langle p_2, \sigma_2 \rangle
\]

These configuration terms may have variables, in which case

\[
\mathcal{R}_{RTL} \vdash (\forall \vec{x}) \langle p_1, \sigma_1 \rangle (\vec{x}) \longrightarrow \langle p_2, \sigma_2 \rangle (\vec{x})
\]

corresponds with symbolic simulation.
Rewriting logic is **reflective**, meaning that $\mathcal{R}_{RTL}$ can used as data-level object ($\hat{\mathcal{R}}_{RTL}$) within $\mathcal{R}_{IR}$.

The inference rules defined in $\mathcal{R}_{IR}$ manipulate proofs from $\mathcal{R}_{RTL}$; i.e.,

$$\mathcal{R}_{RTL} \vdash (\forall x)\langle p_1, \sigma_1 \rangle (x) \longrightarrow \langle p_2, \sigma_2 \rangle (x)$$

is now a data-level object.

The generated test gets carried along within a distinguished object, $\rho$, associated with each proof:

$$\langle p_1, \sigma_1 \rangle (x) \overset{\rho}{\sim} \langle p_2, \sigma_2 \rangle (y) \equiv \rho(\langle p_1, \sigma_1 \rangle (x)) \longrightarrow \langle p_2, \sigma_2 \rangle (y)$$
The inference rules provide the basic "API" through which symbolic simulations are manipulated.

- **ID**: $t \xrightarrow{\text{id}} t$
- **T**: String together two simulations. $t \xrightarrow{\rho_1} t' \quad t' \xrightarrow{\rho_2} t'' \quad t \xrightarrow{\rho_2 \circ \rho_1} t''$
- **I**: Partially-concretize a symbolic simulation. $t \xrightarrow{\rho} t' \quad t \xrightarrow{\rho' \circ \rho} \rho'(t')$
- **RW**: Step the symbolic simulation. $t \xrightarrow{\rho} t' \quad t' \xrightarrow{1} t'' \quad t \xrightarrow{\rho} t''$
Example (The “ID” Rule)

\[ \langle p(x), \sigma \rangle \xrightarrow{\text{id}} \langle p(x), \sigma \rangle = \]
\begin{verbatim}
< always @(posedge clk)
    if ( i )
        count <= count + 1;
\end{verbatim}

\[ x // Testbench (TBD)
, [(clk,0),(i,0)
  ,(count,0)] >

Example (The “ID” Rule)

\[ \langle p(x), \sigma \rangle \xrightarrow{id} \langle p(x), \sigma \rangle = \]
\[ < \text{always } @(\text{posedge } clk) \]
\[ \text{if ( } i \text{ )} \]
\[ \text{count } <= \text{ count } + 1; \]
\[ x \text{ // Testbench (TBD)} \]
\[ , [(\text{clk,0}), (i,0)] \]
\[ ,(\text{count,0})] \]

Example (The “I” Rule)

\[ \langle p(x), \sigma \rangle \xrightarrow{\rho} \rho(\langle p(x), \sigma \rangle) = \]
\[ < \text{always } @(\text{posedge } clk) \]
\[ \text{if ( } i \text{ )} \]
\[ \text{count } <= \text{ count } + 1; \]
\[ \text{initial begin} \]
\[ \#1 i = y; \]
\[ \#5 \text{ clk = 1;} \]
\[ \text{end} \]
\[ , [(\text{clk,0}), (i,0)] \]
\[ ,(\text{count,0})] \]
Example (The “I” Rule)

\[ \langle p(x), \sigma \rangle \xrightarrow{\rho} \rho(\langle p(x), \sigma \rangle) = \langle \text{always } @(\text{posedge } clk) \text{ if ( i )} \text{ count } \leq \text{ count } + 1; \rangle \]

initial begin
  #1 i = y;
  #5 clk = 1;
end
$
\langle (\text{clk,0),(i,0)}, (\text{count,0}) \rangle
$

Example (The “RW” Rule)

\[ \rho(\langle p(x), \sigma \rangle) \rightarrow \langle p', \sigma' \rangle(y) = \langle \text{always } @(\text{posedge } clk) \text{ if ( i )} \text{ count } \leq \text{ count } + 1; \rangle \]

initial begin
  #4 clk = 1;
end
$
\langle (\text{clk,0),(i,y)}, (\text{count,0}) \rangle
$

\
Example (The “RW” Rule)

\[ \rho(p(x), \sigma) \rightarrow (p', \sigma')(y) = \]
\[ \langle \text{always @(posedge clk)} \]
\[ \text{if (i) count <= count + 1;} \]
\[ \text{initial begin}
\[ #4 \text{ clk = 1; end}
\[ , [(\text{clk},0),(i,y) \]
\[ ,(\text{count},0)]\]
\[ > \]

Example (The “RW” Rule)

\[ (p', \sigma')(y) \rightarrow (p'', \sigma'')(y) = \]
\[ \langle \text{always @(posedge clk)} \]
\[ \text{if (i) count <= count + 1;} \]
\[ , [(\text{clk},1),(i,y) \]
\[ ,(\text{count},y ? 1 : 0)]\]
\[ > \]
• Rewriting logic is a computational logic, suitable for declarative programming.
  • Executable through a rewriting logic engine such as Maude.
• We use $\mathcal{R}_{IR}$ as an "API" to manipulate simulations.

**Example**

```plaintext
eq\text{myStrat}(\text{config}) = \text{--- input is } t = \langle p, \sigma \rangle(\vec{x})

let symbSim1 := applyID(\text{config}) \text{ --- } t \xrightarrow{\text{id}} t

rho := ...

symbSim2 := applyI(\rho, symbSim1) \text{ --- } t \xrightarrow{\rho} t'

symbSim3 := applyRW(symbSim2) \text{ --- } t \xrightarrow{\rho} t''

symbSim4 := applyRW(symbSim3) \text{ --- } t \xrightarrow{\rho} t'''

in ... \text{ --- rest of strategy.}
```
A Strategy for Backtracking

Backtracking is accomplished simply by being able to have multiple (first-class) simulation objects in scope at once.

Example

```plaintext
eq stepAndBacktrackIf(symbSim) =
  let symbSim' := applyRW(symbSim)
  in if backtrackCond(symbSim')
    then symbSim
    else symbSim' fi .
```
Efficient solvers can resolve tedious calculations automatically. E.g., an SMT solver for the theory of bit-vectors.

\[ \text{solve} : \text{Formula Judgment} \rightarrow \text{Judgment} \]

Example

```plaintext
solve(\(\varphi\),
< always @(posedge clk)
  if ( i )
  count <= count + 1;
, [(clk,1),(i,y)
  ,(count,y ? 1 : 0)]
>)
```

1. dump \(\sigma\) to a formula:
   \[ \varphi_1 \equiv \text{clk} = 1 \land i = y \land y \rightarrow \text{count} = 1 \land \neg y \rightarrow \text{count} = 0 \]

2. call the solver on \(\varphi_1 \land \varphi\)

3. if assignment \(\rho\) is returned:
   applyI(\(\rho\), < ... >)

4. if not sat., return original judgment.
Verilog Symbolic Interpreter (VSI)

- VSI is a tool supporting the kind of directed testing described above; specifically, for Verilog.
- Sort-of works right now. Has many useful strategies built-in.

Example

```haskell
strat :: Int -> Eval ()
strat 0 = lift $ putStrLn "no solution"
strat j = do
  symbX 1
  m <- solve $ mkValid "hit"
  if isJust m
    then lift $ putStrLn "solution found!"
    else strat (j-1)
```
Some interesting testing conditions for a microprocessor:

- “Perform an instruction that sends negative operands to the ALU.”
- “Perform a load that misses in the cache but hits a victim line being pushed up the memory hierarchy.”

Another interesting test condition, for an 802.11 transmitter:

- “Programmatically test all data rates and check to make sure that OFDM symbols are produced within acceptable time bounds.”
Conclusions and Future Work

Conclusion
A practical improvement to contemporary verification engineering practice.

- Simple idea: make *symbolic simulation* a *first-class object* that can be manipulated programmatically.

Future Work
- Build-up the capabilities of VSI, and apply it to substantial case studies.
- Investigate the idea of interactive test generation.
- Investigate the idea of more efficient test generation by re-use of incremental results.
- Combine with other testing schemes, like directed random.
Conclusions and Future Work

Conclusion

A practical improvement to contemporary verification engineering practice.

- Simple idea: make *symbolic simulation* a *first-class object* that can be manipulated programmatically.

Future Work

- Build-up the capabilities of VSI, and apply it to substantial case studies.
- Investigate the idea of interactive test generation.
- Investigate the idea of more efficient test generation by re-use of incremental results.
- Combine with other testing schemes, like directed random.