Directed Testing as Meta-Reasoning about Simulation

Michael Katelman
(joint work with José Meseguer)

University of Illinois at Urbana-Champaign

September 12, 2009

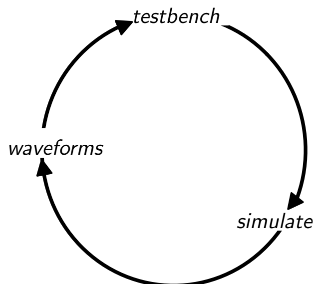
Contemporary Verification Methodology

- 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*

Testing Cycle



- 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.

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
```

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;
...
```

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
```

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;
...
```

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
```

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;
...
```

A Straightforward Solution

- Everyone knows the programmatic solution; *backtracking!*

```
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!*

A Straightforward Solution

- Everyone knows the programmatic solution; *backtracking!*

```
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?

Outline

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.

Outline

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.

Outline

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.

Outline

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.

Outline

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.

The Testing Language By Way of Analogy

Formally, our language is designed so that testing becomes an exercise in *meta-reasoning* about *symbolic simulation*.

Testing Language (Rewriting Logic)	ML & LCF
$\mathcal{R}_{RTL}$ (axiomatization of the RTL semantics)	Any set of axioms in LCF.
$\mathcal{R}_{IR}$ (axioms about $\hat{\mathcal{R}}_{RTL}$)	The inference rules of LCF.
$\mathcal{R}_{STRAT}$ (A testing strategy. $\mathcal{R}_{IR} \subseteq \mathcal{R}_{STRAT}$)	An ML program.

- $\mathcal{R}_{IR}$ and Rewriting Logic constitute "the language" ($\approx$ML).

$\mathcal{R}_{RTL}$

$\mathcal{R}_{RTL}$ 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*.

$\mathcal{R}_{IR}$

- 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.,

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

  is now a data-level object.
- The generated test gets carried along within a distinguished object, ρ, associated with each proof:

$$\langle p_1, \sigma_1 \rangle(\vec{x}) \overset{\rho}{\leadsto} \langle p_2, \sigma_2 \rangle(\vec{y}) \equiv \rho(\langle p_1, \sigma_1 \rangle(\vec{x})) \longrightarrow \langle p_2, \sigma_2 \rangle(\vec{y})$$

$\mathcal{R}_{IR}$

The inference rules provide the basic *"API"* through which symbolic simulations are manipulated.

Rule	Description
$$\frac{}{t \overset{\mathrm{id}}{\rightsquigarrow} t}\ \textbf{ID}$$	Bootstrap.
$$\frac{t \overset{\rho_1}{\rightsquigarrow} t' \qquad t' \overset{\rho_2}{\rightsquigarrow} t''}{t \overset{\rho_2 \circ \rho_1}{\rightsquigarrow} t''}\ \textbf{T}$$	String together two simulations.
$$\frac{t \overset{\rho}{\rightsquigarrow} t'}{t \overset{\rho' \circ \rho}{\rightsquigarrow} \rho'(t')}\ \textbf{I}$$	Partially-concretize a symbolic simulation.
$$\frac{t \overset{\rho}{\rightsquigarrow} t' \qquad t' \longrightarrow^{1} t''}{t \overset{\rho}{\rightsquigarrow} t''}\ \textbf{RW}$$	Step the symbolic simulation.

Detailed Example

Example (The "ID" Rule)

$\langle p(x), \sigma \rangle \stackrel{\text{id}}{\rightsquigarrow} \langle p(x), \sigma \rangle$ =

```
< always @(posedge clk)
    if ( i )
     count <= count + 1;

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

Detailed Example

Example (The "ID" Rule)

$\langle p(x), \sigma \rangle \overset{\text{id}}{\rightsquigarrow} \langle p(x), \sigma \rangle$ =

```
< always @(posedge clk)
    if ( i )
     count <= count + 1;

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

Example (The "I" Rule)

$\langle p(x), \sigma \rangle \overset{\rho}{\rightsquigarrow} \rho(\langle p(x), \sigma \rangle)$ =

```
< always @(posedge clk)
    if ( i )
     count <= count + 1;

   initial begin
    #1 i   = y;
    #5 clk = 1;
   end
   , [(clk,0),(i,0)
     ,(count,0)]
>
```

Detailed Example

Example (The "I" Rule)

$\langle p(x), \sigma \rangle \overset{\rho}{\rightsquigarrow} \rho(\langle p(x), \sigma \rangle) =$

```
< always @(posedge clk)
   if ( i )
    count <= count + 1;

  initial begin
   #1 i   = y;
   #5 clk = 1;
  end
  , [(clk,0),(i,0)
    ,(count,0)]
>
```

Example (The "RW" Rule)

$\rho(\langle p(x), \sigma \rangle) \longrightarrow \langle p', \sigma' \rangle(y) =$

```
< always @(posedge clk)
   if ( i )
    count <= count + 1;

  initial begin
   #4 clk = 1;
  end
  , [(clk,0),(i,y)
    ,(count,0)]
>
```

Detailed Example

Example (The "RW" Rule)

$\rho(\langle p(x), \sigma \rangle) \longrightarrow \langle p', \sigma' \rangle(y)$ =

```
< always @(posedge clk)
    if ( i )
     count <= count + 1;

  initial begin
   #4 clk = 1;
  end
  , [(clk,0),(i,y)
    ,(count,0)]
>
```

Example (The "RW" Rule)

$\langle p', \sigma' \rangle(y) \longrightarrow \langle p'', \sigma'' \rangle(y)$ =

```
< always @(posedge clk)
    if ( i )
     count <= count + 1;

   , [(clk,1),(i,y)
     ,(count,y ? 1 : 0)]
>
```

$\mathcal{R}_{STRAT}$

- 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

```
eq myStrat(config) = --- input is t = ⟨p,σ⟩(x⃗)
  let symbSim1 := applyID(config)         --- t ~id~> t
      rho      := ...
      symbSim2 := applyI(rho,symbSim1) --- t ~ρ~> t'
      symbSim3 := applyRW(symbSim2)       --- t ~ρ~> t''
      symbSim4 := applyRW(symbSim3)       --- t ~ρ~> t'''
  in ... ---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

```
eq stepAndBacktrackIf(symbSim) =
  let symbSim' := applyRW(symbSim)
  in if backtrackCond(symbSim')
      then symbSim
      else symbSim' fi .
```

A Strategy for Automated Solving of Symbolic Conditions

- Efficient *solvers* can resolve tedious calculations *automatically*.
  - E.g., an SMT solver for the *theory of bit-vectors*.
- `solve : Formula Judgment -> Judgment`

Example

```
solve(φ,
 < always @(posedge clk)
    if ( i )
     count <= count + 1;

   , [(clk,1),(i,y)
     ,(count,y ? 1 : 0)]
 >)
```

1. dump σ to a formula:
   $$\varphi_1 \equiv \texttt{clk} = 1 \wedge \texttt{i} = y$$
   $$\wedge\, y \rightarrow \texttt{count} = 1$$
   $$\wedge\, \neg y \rightarrow \texttt{count} = 0$$
2. call the solver on $\varphi_1 \wedge \varphi$
3. if assignment ρ is returned:
   `applyI(ρ, < ... >)`
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

```
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 Real-life Applications

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.