Double the Return from your Property Portfolio:  
Reuse of Verification Assets from Formal to Simulation  

Jonathan Bromley  
Verilab Ltd, Edinburgh, Scotland
Outline

Background:

• Recent massive and welcome increase in formal adoption
• User base no longer limited to specialists with math background

This paper:

If \( w \) is a word over \( \Sigma \), define \( \overline{w} \) to be the word obtained from \( w \) by interchanging \( \top \) with \( \bot \). More precisely,

\[
\overline{w}^i = \top \text{ if } w^i = \bot; \quad \overline{w}^i = \bot \text{ if } w^i = \top;
\]

and \( \overline{w}^i = w^i \) if \( w^i \) is an element in \( 2^P \).

Tools do a great job, but guidelines needed
Scope

• Project: bus interconnect
  – standard protocols on boundaries
  – internal interfaces mostly ad-hoc

• Verification strategy:
  – top-level constrained random TB
  – formal model checking for all internal blocks

• Expectation: **block-level formal TBs remain in place for simulation**

• This paper explores why, how, benefits, problems
SVA enables multi-tool, multi-mode

• SVA is *the same language* for formal and simulation

• In principle, assertions are portable
  – across vendor tools
  – between verification modalities

• Some minor portability issues encountered, but...

**The big problems:**
inherent differences of approach
between formal and simulation
Using assertions in formal and sim

- **Formal TB** written as a module
  - ports match the DUT module
- Bind formal TB **into the DUT**
  - simplifies parameterization

```verilog
module DUT #(parameter A = ...)
  (input ... , output ...);
...
endmodule

module dut_sva_TB #(parameter A = ...)
  (input ... , input ...);
...
  ast_DUT_xxxx: assert property ....
...
endmodule

bind DUT
dut_sva_TB #(A(A), ...)
  inst_TB (*.);
```

trivially easy to include formal TB in simulation
Surely it's unnecessary?

How can simulation of a formal TB be more effective than the formal TB itself?

- **Incomplete exploration** in formal - especially for deep reordering
  - Simulation can exercise long-running scenarios

- **Assertions from formal TB** are valuable **simulation debug aids**

- **Powerful tool to uncover buggy constraints** (assumptions)
### Formal TB

<table>
<thead>
<tr>
<th></th>
<th>Too strict</th>
<th>Too lax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Assert</strong></td>
<td>Assertion fails on valid DUT behavior</td>
<td>Possible DUT bug not detected</td>
</tr>
<tr>
<td><strong>Assume</strong></td>
<td>Overconstraint - some specified behaviors not exercised</td>
<td>Underconstraint - out-of-spec behaviors exercised, DUT may fail</td>
</tr>
</tbody>
</table>

**DUT**

**Assumptions**

**Assertions**
Reversible FV interface blocks

block A

AtoB

BtoA

block B

module AB_intf_sva #(parameter A_is_DUT=1) ...;
...
property B_responds;
  AtoB_req |-> #[1:3] BtoA_rsp;
endproperty
if (A_is_DUT) begin : test_A_mimic_B
  asm_B_responds: assume property(B_responds);
end else begin : test_B_mimic_A
  ast_B_responds: assert property(B_responds);
end
...

interface formal block

implements assume-guarantee methodology
Limitations of reversible blocks

- Behaviour here
- is affected by traffic here
- and credit returns here
- Interface block doesn't know!
Limitations of reversible blocks

• Ideal for self-contained (protocol) interfaces

• Less satisfactory for multiple interdependent interfaces

• *Conclusion*: some important assumptions/assertions are...
  – not checked by assume-guarantee methodology
  – because not re-used on any other formal TB

In practice, some properties cannot be validated by assume-guarantee in formal
Not a seamless transition

• Many debug issues, not given enough early attention
  – initially, many blocks' formal TBs disabled in sim

• Practical concerns, easily worked around
  – auxiliary logic usually OK
    • formal is *more* restrictive than sim
  – some minor syntax irregularities / laxness in formal tools
    • easy to fix

• Serious problems, needed careful attention across many TBs
  – described in following slides
Error messages

• Formal: typically no error message on assertion fail
  – ample information in tools' CEX visualization

• Simulation logs: minimal information for assertion fail
  – most tools have a special debug mode for good tracing of reasons for assertion fail

• well designed error message in the log file can provide adequate debug
Error messages - sampling

- assertion evaluates using Preponed samples
- pass/fail actions executed in Reactive region

assert property ( @(posedge clock) data_1_clock_delay)
else $display("update did not correctly propagate: result='h%h, expected='h%h", result , $past(data) );

update did not correctly propagate: result='h37, expected='h37
• use $\texttt{sampled}$ to get values in Preponed

```
assert property ( @(posedge clock) data_1_clock_delay)
else $\text{display}("update did not correctly propagate: result='h%h, expected='h%h", 
  $\text{sampled(result)}, \ $\text{past(data)} );
```

update did not correctly propagate: result='h4f, expected='h37
Recommendation

- use of $sampled well described in this book
  - along with many other things

big thank-you!
Global variable sampling problem

• Noncompliant behavior of at least one formal tool:

```verilog
code
module TB(...);
    bit invertedMode, sig, exp;
    function automatic bit isOK(bit value, bit expected);
    return (expected == (value ^ invertedMode));
    endfunction
    checkSig: assert property (@... isOK(sig, exp));
endmodule
```

• Simulators strictly honor LRM, our formal tool did not

strict LRM compliance:
- Global variable should be evaluated in Observed
- Function arguments evaluated in Preponed
Performance concerns

• Traditional rule of thumb: avoid liveness properties in simulation

• Reality is more subtle: avoid assertions that create many long-lived threads

• further example – numerous small SVAs in RTL can be burdensome
Undriven / free variables

```verilog
enum {txnNONE, txnWRITE, txnREAD, txnEVICT, ...} txn;

assume property (@... txn==txnWRITE |-> busWnR && busXfr && !busError && ...);
assume property (@... txn==txnEVICT |-> busWnR && !busXfr && !busError && ...);
...
assert property (@... txn==txnWRITE[*2] |=> ...);
...
```

`ifdef ENVIRONMENT_IS_SIMULATION
    always_comb txn = compute_txn_kind(busWnR, busXfr, busError, ...);
`endif

simulation-only driver

undriven TB variable constrains primary inputs

DUT

Care required to avoid unwanted overconstraint

Jonathan Bromley, Verilab
Unexpected benefits

• Cross-checking:
  – another pair of eyes on the formal TB

• Microarchitectural coverage
  – auxiliary logic and formal covers made it easier to get meaningful low-level coverage *e.g.* reordering counts
  – hard to get accurately without this low-level probing
  – *bind* of formal TBs to RTL blocks made it easy
Summary - technical

• Provide informative **error messages** on assertion fail

• Be aware of possible **semantic inconsistencies**

• Avoid liveness assertions that create numerous **long-lived threads**

• Plan for handling of **undriven TB variables** in simulation
Summary – methodology and project

- Ensure all team members aware of formal-to-simulation issues
- Encourage the whole team to engage with formal testbenches
- Use assume-guarantee in simulation to validate formal assumptions
- Take advantage of low-level simulation coverage from formal TBs

Integrate formal TBs into simulation early
Thanks!

Questions?

jonathan.bromley@verilab.com