Formal Verification in the Real World

Jonathan Bromley  jonathan.bromley@verilab.com
Jason Sprott  jason.sprott@verilab.com
Agenda

• Introduction and motivation
• Formal verification refresher
• Challenges
• Power Users' Toolkit
• Planning and Completion
• Q&A
FORMAL VERIFICATION REFRESHER
Formal Property Checking

- Spec. captured as properties
- Formal methods – no simulation
- Prove properties (e.g. SVA) hold
- Exhaustive state space coverage
- Interactive development/debug
- Some limitations

You can even start work without RTL

Start work without many/any properties
Benefits

• Another view on the specification
  – encourages critical mindset
• Potentially exhaustive
• Finds gnarly bugs
• Useful at a very early stage
  – even when RTL and TB are incomplete
• Focus on design behaviour (not stimulus)
• Focused debug (near-minimal CEX)
Drawbacks

• Difficult to tell if your design is suitable for FV
• Can be costly of compute resources
• Time to closure is hard to predict
• Requires skill in all but the simplest cases
• Results not always easy to interpret

It's still worth it!
FV Planning

• Start with expectations
  – Full/bounded proofs
  – Early bring-up
  – Identify goals, e.g. tricky to verify with simulation

• Analyze the actual design

• Repeatable, maintainable, sign-off auditable

• Closure
  – Feed into the overall verification plan
  – Dovetail with simulation
Closure

• Validate all assumptions
  – Assume-guarantee in FV
  – Run in simulation (property style must be considered)
  – Review (sometimes the only way)

• Validate abstractions
  – Are they "safe", i.e. do not over constrain?

• Gaps between simulation and FV
  – Coverage needs looked at closely
  – Extend confidence in bounded proof with simulation
  – Small focused properties (typically better) can leave gaps
  – Abstractions in FV can leave gaps
Start from the Spec

• Interface specifications:
  – no grant without request...
  – if VALID and not READY, everything should be stable...
  – latency limits
  – eventual response (no starvation)

• Protocol specifications:
  – correct number of beats in burst, valid controls, ...

• End-to-end specifications:
  – transaction integrity, routing
  – transaction ordering
Planning the FV effort

• Interface specifications
  – valid/ready handshake integrity
  – handshake eventually completes

• Protocol specifications
  – no new same-ID write until previous ID acknowledged

• End-to-end specifications
  – write goes to correct downstream port with correct data
  – every write gets acknowledged in order

Per-port checkers

Modelling and checker required

FIFO-like checking
## Formal Apps

### Formal Property Checking

<table>
<thead>
<tr>
<th>Bring-up</th>
<th>Develop</th>
<th>Bounded/Full Proof</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto-Property Generation</td>
<td>Extract implementation detail properties</td>
<td></td>
</tr>
<tr>
<td>Register Access</td>
<td>Generate reset, access policy and functional checks</td>
<td></td>
</tr>
<tr>
<td>End-to-end Checkers</td>
<td>IP for hard to develop checker models, e.g. scoreboards</td>
<td></td>
</tr>
<tr>
<td>Unreachable Analysis</td>
<td>Identify unreachable states and save manual analysis</td>
<td></td>
</tr>
<tr>
<td>Coverage Analysis</td>
<td>Are we done? Results: code, COI, proof, functional</td>
<td></td>
</tr>
<tr>
<td>X Propagation</td>
<td>And so on ... with apps working across the flow</td>
<td></td>
</tr>
<tr>
<td>Clock Domain Crossing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SoC Connectivity</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CHALLENGES
Achieving proof and coverage closure

• Typical user experience:
  – useful CEXs found very quickly
  – as simple bugs are fixed, proof times get longer
  – when RTL and TB are mature, some proofs don’t complete

• Reducing RTL design size (parameterization) can help
  – Data widths can be very small
  – FIFO depths, timeout counts, number of ports...

• Consider temporary constraints:
  – one mode at a time
Other techniques

• Abstraction
  – replace counters, memories etc with abstraction that exhibits critical behaviours without full modelling
  – some tool automation
  – skill and experience required in practice

• Invariants and helper assertions
  – Use already-proven assertions as assumptions to reduce state space (may be automatic in the tool)
  – White-box assertions on internal structures
Confidence (or not) in bounded proofs

• Track counterexample lengths over project
  – proof bound must exceed the largest CEX you’ve seen
• Reason about latency through the design
• Reason about cycles required to fill storage, etc
• Proof bound must exceed length of related covers
• Achieve 100% toggle coverage
• Track achieved bound as a function of tool runtime
  – prioritize effort appropriately
Formal-to-simulation challenges

• 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 problem:
inherent differences of approach
between formal and simulation
Code review

• Code quality, clarity, comments/documentation
  – even more important for a formal TB

• Mapping from spec points to assertions
  – logical justification of how modelling+assertions checks a given requirement in full
  – justification of assumptions

• Parameterization
  – justify your choices of parameters (usually smaller than RTL)

• Sanity cover properties for key use cases

• Good use of formal apps
POWER USERS' TOOLKIT - Agenda

- Counterexample waivers
- RTL parameters
- Specimen-value (symbolic) assertions
- Dealing with time hogs
- Managing complexity
CEXs Hide Bugs!

• Tools will find one example violation if possible
  – no requirement to find every CEX

• Multiple DUT bugs may violate just one assertion
  – the CEX you see may lead you to only one of those bugs

• FIX or WORK AROUND to expose the others
Bug hiding and bug waivers

Real life example

• AHB interconnect expects burst to end on error
• AHB VIP can allow burst to continue after error

Waive it, we know our AHB masters don't do that.

But there was another, real bug violating the same assertion

UNDETECTED BUG

CEX: bad burst at slave
Bug waivers - conclusion

• CEX in final regression run is *unacceptable*
  – explained waiver of CEX is untrustworthy

• Add workaround constraints (assumptions)

• Review and waive the constraints
  – each must reflect a known, specified limitation
  – much less risk of missing an unexplained bug
Parameters

- Each parameterization is new RTL
  - Different internal model
- Need strategies for dealing with parameters
  - Symmetry
  - Test compression
  - Testbench reconfiguration

Formal cannot reason about parameters
Parameters - symmetry

• Some parameters are *quantitative*
  – bus width
  – number of ports
  – FIFO depth

• Regular structures

• Examine RTL to find risk areas (structure change):
  – 0, 1, 2^N, 2^N-1, ...

• Otherwise, appeal to symmetry
Parameters – test compression

• Use **pairwise** to manage large parameter spaces
• Prioritize known customer configurations
• Large configurations prove more slowly
  – use small configs for bringup, debug
Parameters – convert to signals?

• Can the parameter be reworked as a pin-strap option?
  – priority weights, address maps, count limits, ...

• Use arbitrary rigid values, not parameters
  – Assumptions to enforce legal values

• Formal tests every possibility

It's an RTL transformation – needs checking
Testbench reconfiguration

• Some parameter values make coverage unreachable
• Example:
  – AHB bus width: \textit{transfer size} > \textit{bus width} is impossible
• Waiving missing coverage: \textit{messy, laborious}
• Aggregate coverage across parameterizations: \textit{weak}

Use generate-loops to remove unreachable covers and asserts?
Safer testbench reconfiguration

• Generate and exclude directives as appropriate...

• ...but check your exclusion is safe using a generated assertion!

```verilog
if (DATA_WIDTH >= 64) begin : gen_doubleword
    ast_dword: assert property (
        (HTRANS == AHB_TRANS_NONSEQ) &&
        (HSIZE == AHB_SIZE_DWORD)
    |-> ........
    );
end
else begin : gen_no_doubleword
    ast_never_dword: assert property (
        HTRANS == AHB_TRANS_NONSEQ
    |->
        HSIZE != AHB_SIZE_DWORD
    );
end
```

• Protects against coding goofs
• Better checking of DUT
Specimen values

• Some attribute \( A \) (address, ID, mode, ...)

• Prove something for an \textit{arbitrary} value of \( A \) ...

• ... then it is proven for \textit{all possible} values of \( A \)
  – because proof must consider all possible arbitrary values
  – doesn't prevent other values appearing as well

• Compared with local property variables:
  – specimen transactions can be more efficient
  – usually easier to understand and write
  – value is available across multiple properties

Sometimes called \textit{symbolic checking}

Local property variables may be better in simulation
Example of specimen value

• Pending-transaction counter for a specimen ID value

\[
\text{assume property ( } \text{\$stable(specimen_id)} \text{ );}
\]
\[
\text{assign push = w_hsk && (w_id == specimen_id);}
\]
\[
\text{assign pop = b_hsk && (b_id == specimen_id);}
\]
\[
\text{always @(posedge clock or negedge areset_n)}
\]
\[
\text{if (~areset_n)}
\]
\[
\text{id_in_flight <= 0;}
\]
\[
\text{else}
\]
\[
\text{id_in_flight <= id_in_flight + push - pop;}
\]

• "Never two same-ID transactions in flight" is now easy:

\[
\text{assert property ( id_in_flight <= 1 );}
\]

Pick an arbitrary ID to observe

Sample-ID proof is sufficient for all IDs
Performance gains

• Specimen assertions slower to prove:
  – about 2x or 3x slower than fixed-value assertions

• but **one** specimen assertion works for all possible values!
  – tools exploiting symmetry?

Massive improvement when many values must be checked

In real-world example
Surprising example

• Design has $N_M$ master ports, $N_S$ slave ports
• Check all $N_M \times N_S$ master-slave paths?
  – Very poor runtime
  – Minimal verification value (symmetry)
• Use specimen master and slave numbers $M, S$
• Check only the path from master $M$ to slave $S$
• Better tool runtime, no loss of verification power
  – Tools exploit symmetry automatically where possible
Time-hog hotspots

• Simulation runtime is highly predictable
  – a few minutes of sim gives good estimate of sim speed in clock cycles/sec

• Formal much less so
  – if an assertion is slow to prove, it's almost impossible to predict how much longer it will take
Suggestions (1): Scaling

• Start with very small parameterization
  – find time for complete proof

• Progressively increase parameterization size
  – find how proof time scales with increasing size
    • linear? quadratic? exponential?
  – helps to estimate proof time for realistic sizes
Suggestions (2): Isolation

• Prove just one or two problematic assertions
  – is their proof time reasonable?
  – if so, maybe more compute power is the answer

• but remember to run all assertions for a while first!
  – provides intermediate results, may make difficult proofs go faster
Suggestions (3): Bounded Proofs

• Observe proof radius (number of cycles explored)
• Make reasoned decisions about validity of bounded proof
  – Longer than any CEXs seen during debug
  – Reasoning about expected duration of activity
• Many good papers and other references available
Suggestions (4): Skip redundant checks

• AHB SEQ transfer must have correct relationship to previous SEQ or NONSEQ transfer
  – same HSIZE, HPROT etc
  – correctly incremented HADDR

• Assumptions enforce correct behaviour by master
  – essential, but minimal performance impact

• Assertion check at slave is **slow**
  – and **unnecessary**

Other checks cover:
  • transaction ordering
  • transaction integrity
Managing Complexity

• Design+Properties = one huge state machine!
• More complexity $\rightarrow$ slower proof

Reduce complexity associated with slow proofs

• Well established techniques:
  – black-boxing
  – cutpoints
  – abstraction
Simplify Unnecessary Difficult Stuff

- Big complex blob of logic in the cone-of-influence of CHECK
  - Slow proof
  - CHECK is not really trying to test B

Examples:
- big math function
- long time delay
Simplify by Black-boxing

- Remove the logic altogether
- Standard option in formal tools
- Default for some blocks (multiplier...)

These signals are now free
Simplify using Cutpoints (stop-at)

- Removes complexity from selected signal paths

ENV: assume property (...);

DUT

A → B → C

CHECK: assert property (...);

This signal is now free
Why is it OK to destroy the design?

• Verification-only, within formal tool
  – No impact on RTL integrity
• If assertion proves with a signal free, then it is also proven for the "correct" values
• Verification is more rigorous than without the cutpoint or black-box!
Verification across Cutpoints

- B functionality can be tested by another assertion

This signal is now free
Abstraction

- Replace real logic with massively simplified model

Examples:
- long timeout counter
- handshake eventually
Built-in abstractions

• Tools provide ready-to-use counter abstractions:
  – reset
  – limits
  – "critical values"

• And maybe some arithmetic

• etc
Your own counter abstractions

• It's not so hard:

  assume property (s_eventually count == LIMIT);

• Add custom features to avoid crazy behaviour:

  assume property (start_timeout |=> (count != LIMIT) [*2];);

  no timeout for at least 2 cycles after trigger

  assume property (count == LIMIT |-> $past(count == LIMIT-1);

  partial count sequence

other RTL may need certain specific counter behaviours
PLANNING AND COMPLETION
Planning for low stress

• Give the engineers time to experiment and gain experience
  – Poor early decisions, frozen because of timescale pressure, can be hugely counterproductive

• Encourage teamwork and discussion
  – This stuff is not intuitive
  – Single engineer can easily become stalled

• Find internal expertise
  – and make sure it's accessible
Planning for completeness

• Start with the spec
• Demand clear justification of how each spec point is verified by one or more assertions
  – Relationship isn't one-to-one
  – Justification may be complex, needs thoughtful review
  – Don't tolerate BS
  – Don't encourage BS
Validating completeness

• When are you done?
• FV effort is only as good as your spec points

But the tools can provide some quality metrics

• Code coverage / reachability
• Cone of influence
  – and "proof core"
"Code" coverage

• As in simulation:
  – code coverage as *sine qua non*
  – helps identify weaknesses in TB
  – can never be functionally exhaustive
  – may need elaborate waivers

Expect formal coverage to be smarter than simulation code coverage
Cone of influence

• Establish which parts of the RTL are checked by each property
  – Can occasionally highlight missing checks
  – Typically, design complexity means more logic appearing in COI than you care about

• Vendors have intelligent variants on COI
  – "proof core" or similar
  – results often very hard to interpret, need support from vendors
Conclusion: Real People Still Required

• Blind faith in tool reports is a good way to miss bugs
• No substitute for a bunch of smart engineers thinking it through
• Mismatch between abstraction levels for FV and other forms of coverage
  – makes review process much harder
  – deep-dive often required
  – allow time for it!
RESOURCES

• Tool vendors are helpful
  – genuinely expert field engineers
  – knowledge-base websites

• More information related to this tutorial:
  http://www.verilab.com/resources/
  – follow "formal tutorials" link
  – many internal and external links
Q&A