Formal Verification: Too Good to Miss

Jonathan Bromley, Jason Sprott
Duration: 15 minutes

SECTION 1:
PROMISE AND PLANNING
Formal Property Checking

You can even start work without RTL

DUT
RTL

FPC
Prove

Assert
Assume
Cover

result | debug

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

Start work without many/any properties

© Accellera Systems Initiative and Verilab
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!
Use Models – Shift Left

- Find bugs earlier
- Incomplete TB useful
- Improve specification
- Value from FV done on other blocks, e.g. assume-guarantee

Detailed bug hunting
- Can begin before RTL and formal TB are complete
- FV finds concise CEX
Use Models – Block Signoff

Formal Property Checking

Bring-up   Develop   Bounded/Full Proof

Thorough checking and proof

- TB and RTL complete
- Rare to get full proofs of everything
- Detailed review of formal TB is vital
- Apps, e.g. auto-generated properties

Late design changes, rogue bug hunting

© Accellera Systems Initiative and Verilab
Save Time Using Formal Apps

**Formal Property Checking**

- Bring-up
- Develop
- Bounded/Full Proof

**Auto-Property Generation**
- Extract implementation detail properties

**Register Access**
- Generate reset, access policy and functional checks

**End-to-end Checkers**
- IP for hard to develop checker models, e.g. scoreboards

**Unreachable Analysis**
- Identify unreachable states and save manual analysis

**Coverage Analysis**
- Are we done? Results: code, COI, proof, functional

**X Propagation**
- And so on … with apps working across the flow

**Clock Domain Crossing**

**SoC Connectivity**

© Accellera Systems Initiative and Verilab
Use Models – Architecture

• Replace the RTL design with “assume property” directives
• Can point to architectural issues:
  – deadlocks
  – latency limits
  – …
The Big Questions

Q1: Can I use FV on this block?

Q2: How do I get started?
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
Start With Expectations

It's not all about full proofs!

Full Proofs

- Bug absence assured
- Difficult to predict if achievable
- Depends on size/type of design
- Takes time

Undetermined

- AKA "bounded"
- Valuable
- Validate depth by analysis?
- Can be quick
Plan For Early Bring-Up

• Essential for initial design analysis
• Designers can start on draft RTL
• Ask and capture simple questions
• Early/often drops, adding/fixing functionality
• Look for collateral
  – Interface and protocol assertions
  – Embedded properties
  – FV end-to-end checkers
  – Abstractions, e.g. counters, memories
• Results useful down the line
Analyze: Complexity, States, Depth

module response_reorder #(
    // Number of downstream channels
    N_CHANS = 4,
    // Bit-width of data and ID signals
    BITS_DATA = 2,
    BITS_ID = 3,
    // Maximum number of un-acknowledged
    // (depth of reorder buffer)
    DEPTH = 6
)

Are all channels identical?
Are all bits treated the same way?
Can we simplify by modeling?

© Accellera Systems Initiative and Verilab
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
SECTION 2: DELIVERING

Duration: 45 min
Start from the Spec

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

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

• End-to-end specifications:
  – transaction integrity, routing
  – transaction ordering
Analyze The Design?

• Structure
  – Partitioning, size (e.g. I/O, flops)
• Functionality
  – FSMs, counters, arithmetic, memory
  – States, sequential depth
  – Clocks, resets, asynchronous logic
• Properties (assert, assume, cover)
  – Protocols, embedded, assume-guarantee
• Parameterization
  – No shortcuts using FV
• Leverage
  – Symmetry, abstractions
An example

• Response steering/ordering block
• Loosely based on AXI AW-to-B channel relationships
• Massively simplified
  – for tutorial clarity
  – to avoid any confidentiality issues
Example DUT

**response_reorder**

**INITIATOR**
- w_chan
- w_valid
- w_ready
- w_data
- w_id

**RESPONDER**
- b_valid
- b_ready
- b_id

**SLAVE 0**
- s_valid[0]
- s_ready[0]
- s_data[0]

**SLAVE N-1**
- s_valid[N-1]
- s_ready[N-1]
- s_data[N-1]
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
Port protocol

- AXI-style valid/ready handshake
- Transfer when (VALID && READY)
  - Payload and VALID must remain stable until READY
End-to-end specifications

• Responses strictly in initiator order
  – but no limits on overlapping of downstream activity

• Response cannot be given until transaction has completed downstream

• No two same-ID transactions in flight together
  – strict write observability

• Data exactly preserved from initiator to selected slave

• Initiator ordering preserved at each slave
From spec to testbench

- Responses strictly in initiator order
  - but no limits on overlapping of downstream activity
- Response cannot be given until transaction has completed downstream
- No two same-ID transactions in flight together
  - strict write observability
- Data exactly preserved from initiator to slave
- Initiator ordering preserved at each slave

- Don't think about stimulus!
  - Focus on spec rules
- A surprisingly short list of key rules may be enough
  - protocol on ports
  - ordering among transactions
  - ordering between ports
  - transaction integrity
Coding the properties

- **Assertion** checks correct behaviour on DUT outputs
- **Assumption** enforces correct behaviour on inputs
  - constrains TB to follow valid protocol

```verilog
property p_payload_stable(VAILD, READY, PAYLOAD);
disable iff (~areset_n) @(posedge clock)
  VALID && !READY |=> $stable(PAYLOAD);
endproperty

ast_b_id_stable: assert property (p_payload_stable(b_valid, b_ready, b_id));

asm_w_id_stable: assume property (p_payload_stable(w_valid, w_ready, w_id));
```
Inside-out testbench structure

- **Formal TB** is a module
  - ports match **DUT module**
- **Bind TB** into the **DUT**

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

... 
endmodule
```

```verilog
module dut_sva_TB #(parameter A = ...)
    (input ... , input ...);

... 
ast_DUT_xxxx: assert property .... 
...
endmodule
```

```verilog
bind DUT
dut_sva_TB #(A(A), ...) bound_TB (.*);
```
Testbench structure

• Formal TB has *only input ports*
  – Input ports are free to take any value at any time ...
  – ... assumptions act as constraints, not drivers

• Inside-out structure *unaltered* works in simulation
  – assumptions act as assertions

• FV tools allow for *scripted* addition of
  – properties
  – stopats
  – modules
  – ...
Unbounded latency

- Use in assumptions to guarantee forward progress
- Use in assertions to check no lockup
- Beware potential inefficiency / weakness in simulation

Liveness property

```
property p_eventually_ready (VALID, READY);
  disable iff (~areset_n) @(posedge clock)
  VALID |-> strong (##[0:$] READY);
endproperty
```

Also SV-2009 eventually

SV-2009 keyword, required for some tools
Finding a real DUT bug

• Flawed early version of DUT can lose some responses

• Found in less than 1 sec runtime!

Failure shown in assertions list

Looping trace shows failure of liveness property
Finding a testbench bug

- Flawed assertion
  “no duplicate ID in flight”

```plaintext
sample_written && !sample_responded |
(!w_hsk && (w_id == sample_id))) ||
(b_hsk && (w_id == sample_id))
```

- Fix assertion to allow same-ID W and B in the same clock cycle
Assume-guarantee

- Powerful check on the properties themselves
  - Flaws will cause false failures in one or other case
  - Easily automated with parameters/generates

```c
assert property ( 
p_payload_stable(  
s_valid[0],  
s_ready[0],  
s_data[0])  
);  

assume property ( 
p_payload_stable(  
valid,  
ready,  
data)  
);  
```
Finding flawed properties

- Cross-check
  - by assume-guarantee
  - or by simulation

- Review goals

<table>
<thead>
<tr>
<th></th>
<th>Too strict</th>
<th>Too lax</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert</td>
<td>False fails</td>
<td>DUT errors missed</td>
</tr>
<tr>
<td>assume</td>
<td>Some behaviours not exercised</td>
<td>DUT fails because of bad input</td>
</tr>
<tr>
<td>cover</td>
<td>Unreachable covers</td>
<td>Coverage not trustworthy</td>
</tr>
</tbody>
</table>

© Accellera Systems Initiative and Verilab
Free variables, arbitrary tie-offs

• Default: undriven signals and inputs are free
  – Can take any value at any time
  – Constraints (assumptions) needed to enforce protocol

```
asm_fixed_but_arbitrary_cfg:
  assume property (    
    disable iff (~areset_n) @(posedge clock)     
    $stable(cfg)                                      
  );
```

• Useful idiom: a free variable that's held constant
  – proof must take account of all possible values
  – but value doesn't change from one clock to the next
Another useful modelling idiom

• Pulse for exactly one clock, just once, at an arbitrary time during each trace – don't try to simulate this!

```vhdl
assume property ( disable iff (~areset_n) @(posedge clock) pulse ##[1:$] |-> !pulse;);
```

Also allows for no pulse, ever
Auxiliary logic (1)

• Combinational encapsulation

```verilog
assign w_hsk = w_valid && w_ready;
```

• Event-has-happened recording

```verilog
always @(posedge clock or negedge areset_n)
  if (~areset_n)
    push_seen <= 0;
  else if (push)
    push_seen <= 1;
```
Specimen transactions

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

• Prove something for an arbitrary value of $A$ ...

• ... then it is proven for 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

Local property variables may be better in simulation
**Auxiliary logic (2)**

- Pending-transaction counter for a specimen ID value

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

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

```verilog
assert property ( id_in_flight <= 1 );
```

Pick an **arbitrary** ID to observe

Sample-ID proof is sufficient for **all** IDs
Ordering

- Responses strictly in initiator order
  - but no limits on overlapping of downstream activity
- Response cannot be given until transaction has completed downstream
- No two same-ID transactions in flight together
  - strict write observability
- Data exactly preserved from initiator to slave
- Initiator ordering preserved at each slave

- Full description of ordering:
  - hard to write
  - typically needs lots of auxiliary logic
  - often very inefficient for formal tools

- Find a simplified description that captures all the requirements
FIFO ordering by counting

• Presented by Darbari and Singleton in several 2015 papers

✓ Track how many transactions are in the FIFO
✓ Pick some arbitrary transaction
✓ If track_count == N on push, then this transaction must be the Nth to be popped

References available

push_count – pop_count
specimen transaction

Complete description of FIFO in one simple assertion!
Picking an arbitrary transaction

- Specimen-transaction is not sufficient
  - id==specimen_id will always pick the first with that ID

- Add nondeterminism using a free variable

```verilog
bit sample_accept;
assign push = sample_accept && (w_id == sample_id) && w_hsk;
assume property (sample_written |-> !sample_accept);
```

- Combine with sample-has-happened modelling
  - exactly one specimen transaction
SECTION 3: FOLLOWING THROUGH

Duration: 20 min
**Reminder: Formal Apps**

Formal Property Checking

- **Bring-up**
- **Develop**
- **Bounded/Full Proof**

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

© Accellera Systems Initiative and Verilab
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
Covering multiple parameterizations

• Formal verifies only one parameterization per run
• Regression suite may need many parameterizations
• Consider pairwise or other test compression
• Logical proof of parameter relationships?
• Larger parameterizations are MUCH slower to prove
  – but techniques exist to mitigate this

Some parameters can be converted into pin-strap options
Applying block-level formal TB in subsystem-level simulation

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

• Microarchitectural coverage
  – auxiliary logic and formal covers make it easier to get meaningful low-level coverage \textit{e.g.} reordering counts
  – hard to get accurately without low-level probing
  – \textit{bind} of formal TBs to RTL blocks simplifies this

• Incomplete exploration in formal
  – especially for deep reordering, long FIFOs, timeouts...
  – simulation can exercise long-running scenarios
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
Review process (1): code

• 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
Hidden vacuity gotchas

- Beware multiple options in a single antecedent

```
assert property ( A || B => C );
```

Either A or B sufficient to cover this

```
assert property ( A => C );
assert property ( B => C );
```

Is this what you meant?

- No failure if buggy constraints make B impossible

```
cover property ( A && !B );
cover property ( !A && B );
cover property ( !A && !B );
cover property ( A && B );
```

Confirm that all input combinations are reachable?

- No failure if C is stuck true

```
cover property ( !C );
```
Review process (2): results

• Reproducibility
• All assertion preconditions covered
• All assertion and assumption preconditions covered in simulation
• No CEXs or unreachable covers
• Agreed set of parameterizations
• Justification for acceptability of bounded proofs
• Justification for any removed/waived checks
Duration: 5 min + Q&A

CONCLUSIONS
Is It For Me?

• Think why you need FV
  – Early bring-up and critical thinking alone compelling
  – Features tricky to verify using simulation
  – Look at the types of bugs escaping
  – Could you use apps, e.g. X-propagation?
• A lot depends on your design implementation
• Skills are required
  – Current expertise (using assertions already?)
  – In-house FV experts very useful
  – Ability to learn (it'll take time and effort)
• You can start and get ROI quickly

Enables RTL designers to own more verification
Where to go next

• There are good papers and presentations available

• Formal tools differ – take the vendor training

If you want find out about some useful resources leave us your details or mail us

jonathan.bromley@verilab.com
jason.sprott@verilab.com
Thank You!

Any Questions?