2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Formal Verification in the Real World

Jonathan Bromley jonathan.bromley@verilab.com

Jason Sprott jason.sprott@verilab.com

verilab

Agenda

- Introduction and motivation
- Formal verification refresher
- Challenges
- Jumping the Hurdles
- Planning and Completion
- Q&A

FORMAL VERIFICATION REFRESHER

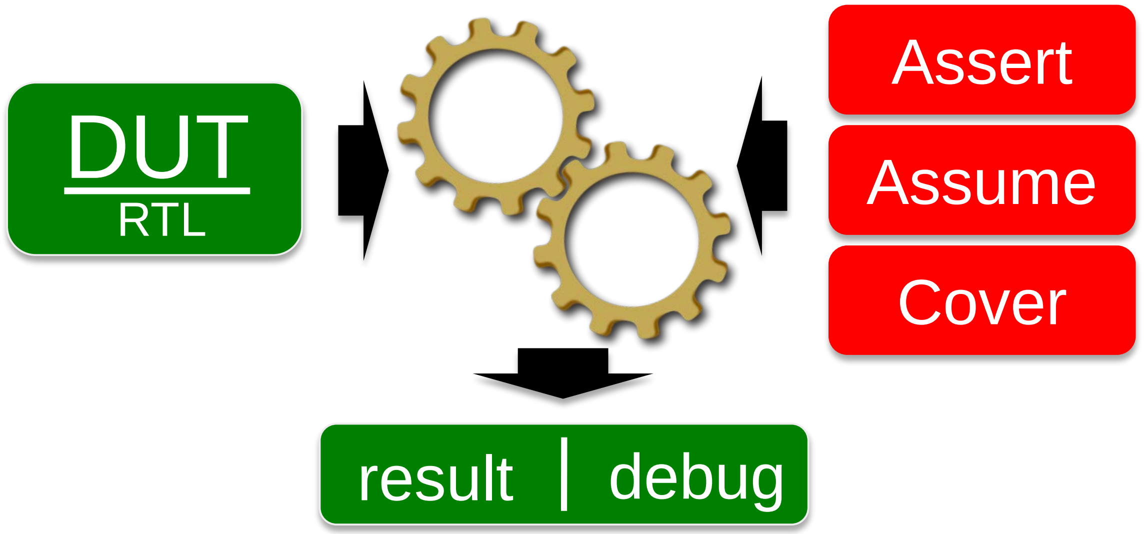
New to formal property checking?

- See our intro and demystifier:

http://www.verilab.com/files/dvcon_eu_2016_fv_tutorial.pdf

- Next few slides: a very quick summary

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

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
  - Early bring-up
  - Identify goals, e.g. tricky to verify with simulation
  - Full/bounded proofs?
- Analyze the actual design
- Repeatable, maintainable, sign-off auditable
- Closure
  - Feed into the overall verification plan
  - Dovetail with simulation

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

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

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
- Clock Domain Crossing
- SoC Connectivity

And so on … with apps working across the flow

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)
  - small data widths
  - 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 (may be automatic in the tool)
  - white-box assertions on internal structures

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 (start with smaller than RTL)
- Sanity cover properties for key use cases
- Good use of formal apps

JUMPING THE HURDLES

JUMPING THE HURDLES - Agenda

- Counterexample waivers
- RTL parameters
- Specimen-value (symbolic) assertions
- Dealing with time hogs
- Managing complexity
  - abstraction, cutpoints, black-boxing

CEXs Hide Bugs!

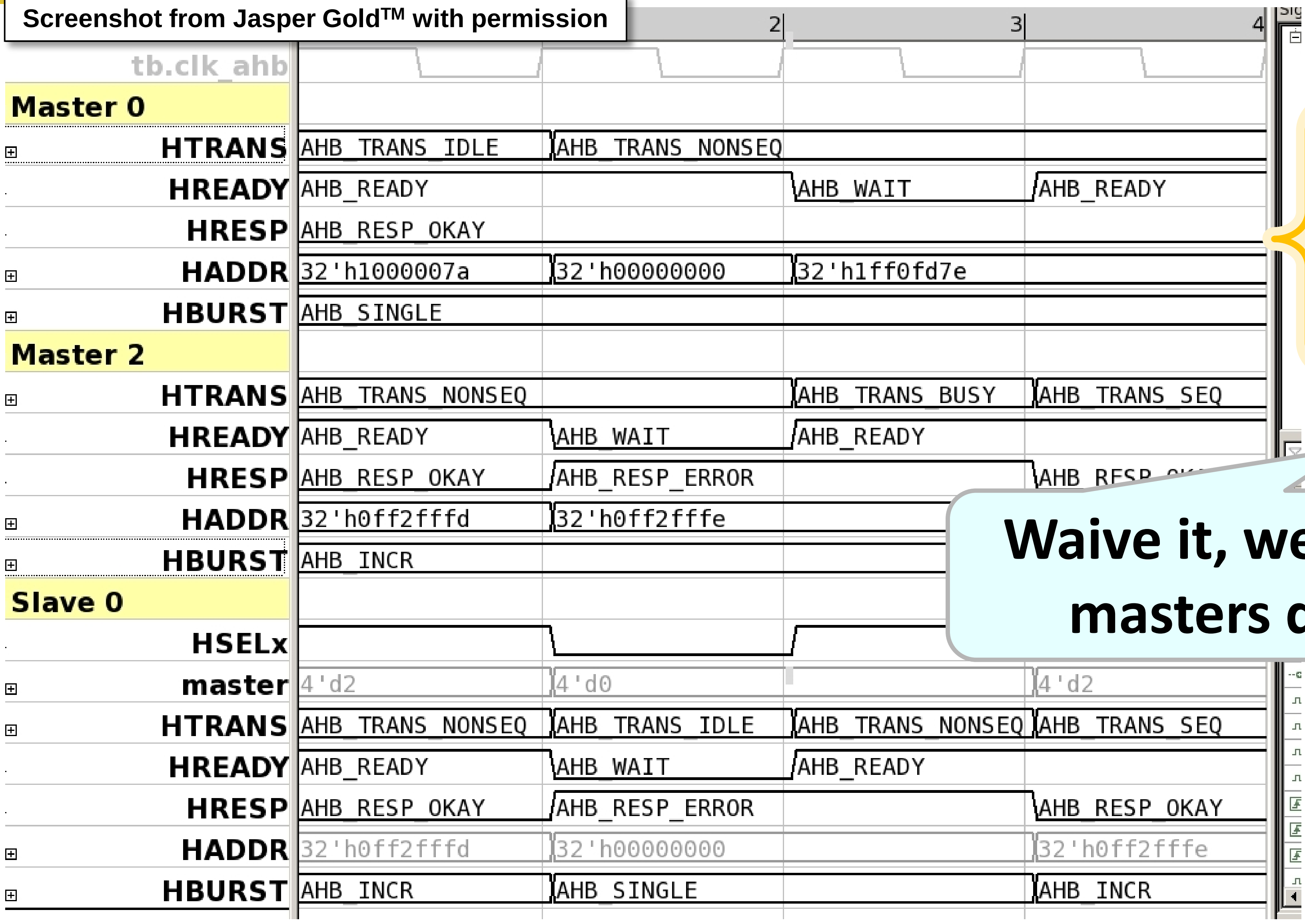
- Tools will find **one example** violation if possible
  - – no requirement to find **every** CEX

- 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

Screenshot from Jasper Gold™ with permission


2
3
4
tb.clk_ahb
Master 0
HTRANS AHB_TRANS_IDLE AHB_TRANS_NONSEQ
HREADY AHB_READY AHB_WAIT AHB_READY
HRESP AHB_RESP_OKAY
HADDR 32'h1000007a 32'h00000000 32'h1ff0fd7e
HBURST AHB_SINGLE
Master 2
HTRANS AHB_TRANS_NONSEQ AHB_TRANS_BUSY AHB_TRANS_SEQ
HREADY AHB_READY AHB_WAIT AHB_READY
HRESP AHB_RESP_OKAY AHB_RESP_ERROR AHB_RESP_OK
HADDR 32'h0ff2fffd 32'h0ff2fffe
HBURST AHB_INCR
Slave 0
HSELx
master 4'd2 4'd0 4'd2
HTRANS AHB_TRANS_NONSEQ AHB_TRANS_IDLE AHB_TRANS_NONSEQ AHB_TRANS_SEQ
HREADY AHB_READY AHB_WAIT AHB_READY
HRESP AHB_RESP_OKAY AHB_RESP_ERROR AHB_RESP_OKAY
HADDR 32'h0ff2fffd 32'h00000000 32'h0ff2fffe
HBURST AHB_INCR AHB_SINGLE AHB_INCR

Real life example

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

CEX: bad burst at slave

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

- **Another bug** violating the same assertion…

UNDETECTED BUG

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

JUMPING THE HURDLES - Agenda

- Counterexample waivers
- RTL parameters
- Specimen-value (symbolic) assertions
- Dealing with time hogs
- Managing complexity
  - abstraction, cutpoints, black-boxing

Parameters

Formal cannot reason about parameters

- Each parameterization is new RTL
  - Different internal model
  - New formal verification run required

- Need strategies for dealing with parameters
  - Symmetry
  - Test compression
  - Testbench reconfiguration

Parameters - symmetry

- Some parameters are *quantitative*
  - bus width
  - number of ports
  - FIFO depth
- Regular structures

```
module myDUT
  #(parameter WIDTH=8)
  (input [WIDTH-1:0] dIn, ...);




  logic [$clog2(WIDTH)-1:0] bitSelect;
  ...
  bitSelect = '1; // select MSB
```

- 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
  - References on our website
- Prioritize known customer configurations
- Large configurations prove more slowly
  - use small configs for bringup, debug

Pairwise example:
5 params, 8 values each

Exhaustive:
8^5 = 32768 tests

Pairwise:
86 tests

Parameters – convert to signals?

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

```
module myDUT
  (input [5:0] TIMEOUT, ...);
  ...
  if (count == TIMEOUT) ...
```

- Use rigid input values, not parameters
  - Assumptions to enforce legal values
- Formal tests every possibility

```
assume property ($stable(TIMEOUT));
assume property (TIMEOUT >= 8);
```

It's an RTL transformation – needs checking/review

Testbench reconfiguration

- Some parameter values make coverage unreachable
- Example:
  - AHB bus width: **transfer size > bus width** is impossible

- Waiving missing coverage: **messy, laborious**
- Aggregate coverage across parameterizations: **weak**

Use generate-blocks 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

```
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**

JUMPING THE HURDLES - Agenda

- Counterexample waivers
- RTL parameters
- Specimen-value (symbolic) assertions
- Dealing with time hogs
- Managing complexity
  - abstraction, cutpoints, black-boxing

Specimen values

Sometimes called **symbolic checking**

- 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

Example of specimen value

- Pending-transaction counter for a specimen ID value

Pick an **arbitrary** ID to observe

```
assume property ( $stable(specimen_id) );

assign id_push = push && (w_id == specimen_id);
assign id_pop  = pop  && (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 + id_push – id_pop;
```

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

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

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

Performance gains

In real-world example

- 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?
- Improves:
  - code clarity
  - coding effort
  - runtime efficiency

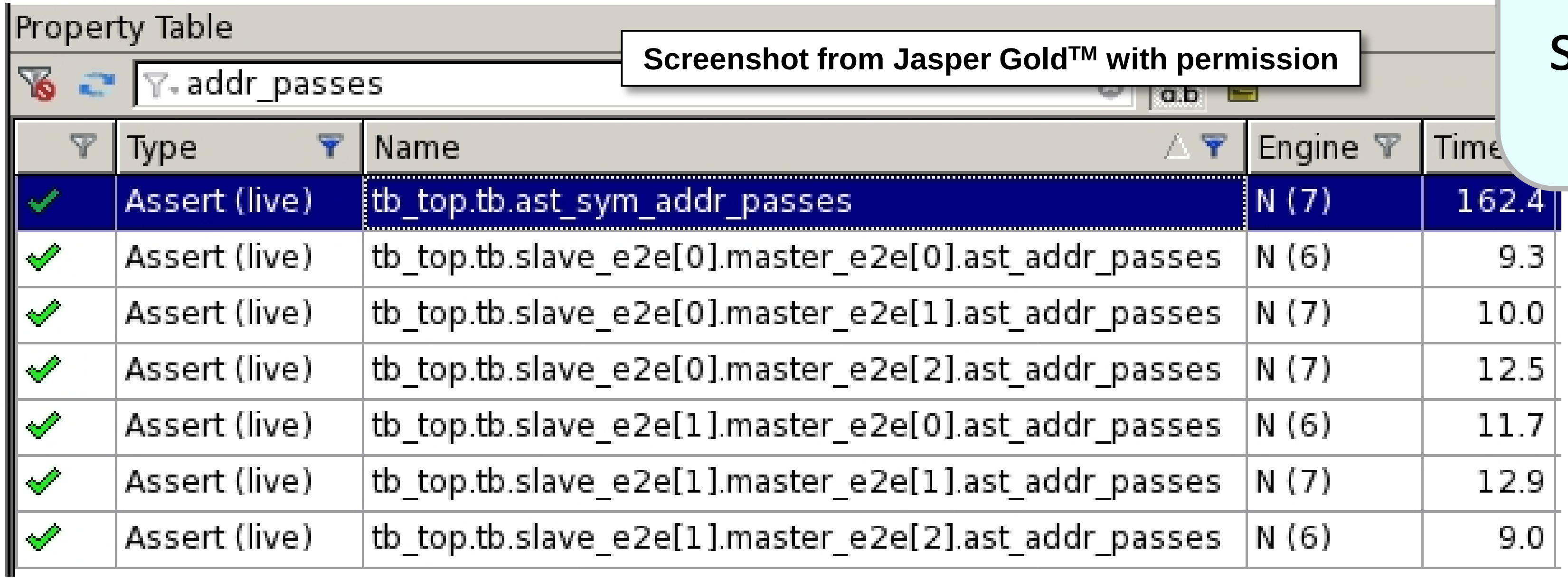
Massive improvement when many values must be checked

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

YMMV example

- Symbolic assertion is not necessarily faster!
- Always sanity-check



Property Table

Screenshot from Jasper Gold™ with permission

addr_passes

	Type	Name	Engine	Time
✓	Assert (live)	tb_top.tb.ast_sym_addr_passes	N (7)	162.4
✓	Assert (live)	tb_top.tb.slave_e2e[0].master_e2e[0].ast_addr_passes	N (6)	9.3
✓	Assert (live)	tb_top.tb.slave_e2e[0].master_e2e[1].ast_addr_passes	N (7)	10.0
✓	Assert (live)	tb_top.tb.slave_e2e[0].master_e2e[2].ast_addr_passes	N (7)	12.5
✓	Assert (live)	tb_top.tb.slave_e2e[1].master_e2e[0].ast_addr_passes	N (6)	11.7
✓	Assert (live)	tb_top.tb.slave_e2e[1].master_e2e[1].ast_addr_passes	N (7)	12.9
✓	Assert (live)	tb_top.tb.slave_e2e[1].master_e2e[2].ast_addr_passes	N (6)	9.0

Proof time for symbolic assertion: **162.4 sec**

Total proof time for exhaustive paths: **65.4 sec**

JUMPING THE HURDLES - Agenda

- Counterexample waivers
- RTL parameters
- Specimen-value (symbolic) assertions
- Dealing with time hogs
- Managing complexity
  – abstraction, cutpoints, black-boxing

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
- Tool coverage app may offer deeper insight
  - what was tested, what wasn't
- Many good papers and other references available

Useful starting point

Kim, Park, Singh, Singhal (DVCon San Jose 2014): Sign-off with Bounded Formal Verification Proofs

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% structural (code) coverage
- Track achieved bound as a function of tool runtime
  – prioritize effort appropriately

JUMPING THE HURDLES - Agenda

- Counterexample waivers
- RTL parameters
- Specimen-value (symbolic) assertions
- Dealing with time hogs
- Managing complexity
  – abstraction, cutpoints, black-boxing

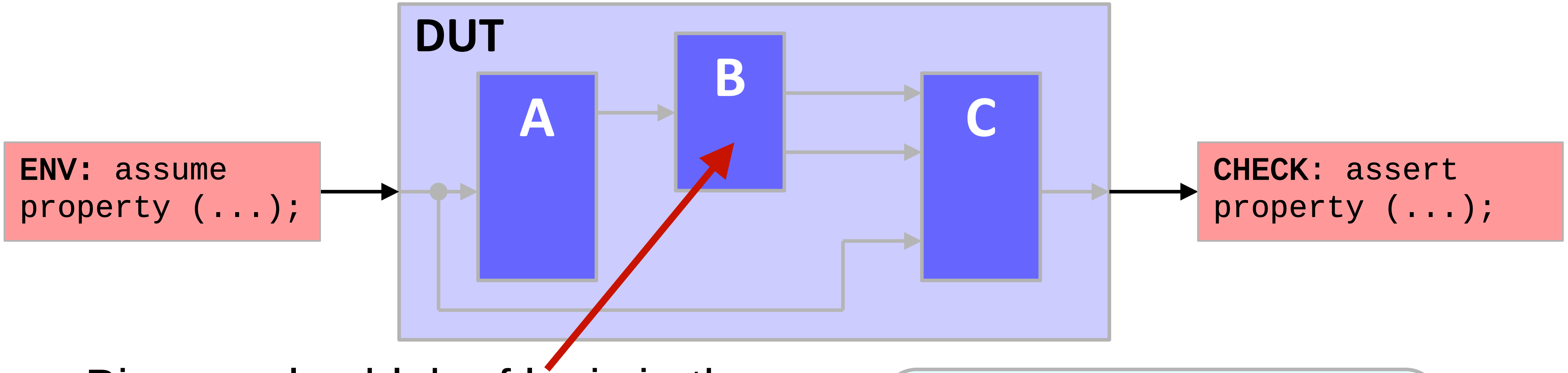
Managing complexity

- Design+Properties = one huge state machine!
- More complexity → slower proof

Reduce complexity associated with slow proofs

- Well established techniques:
  - black-boxing
  - cutpoints
  - abstraction

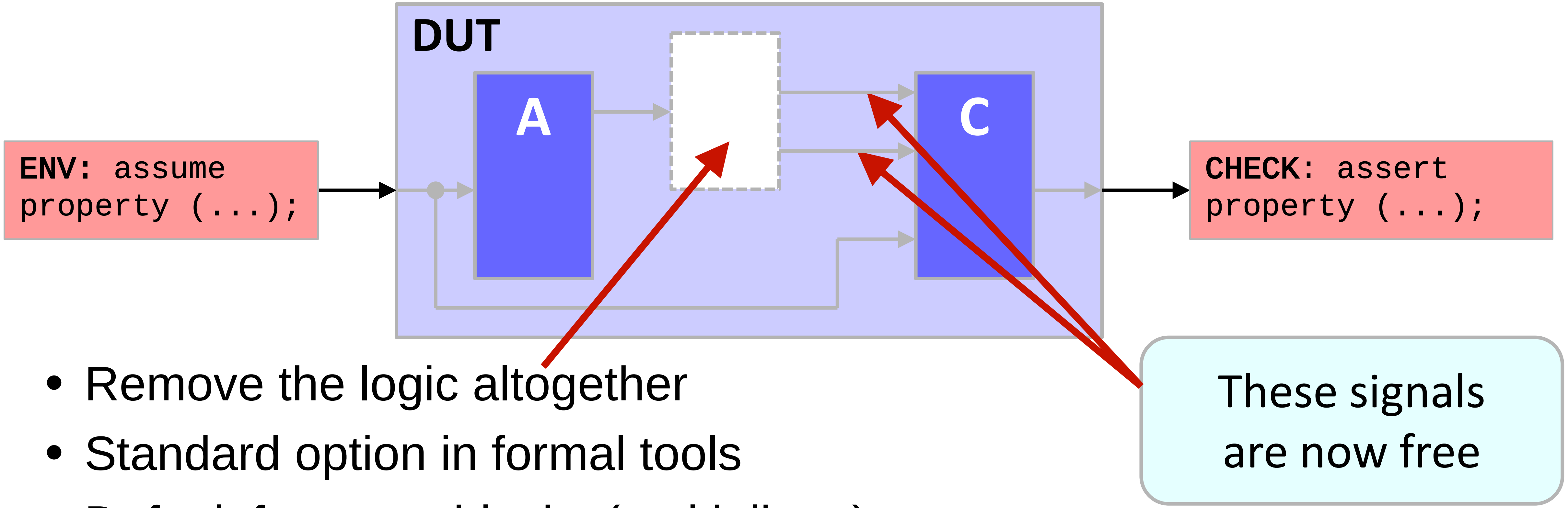
Simplify unnecessary difficult stuff



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

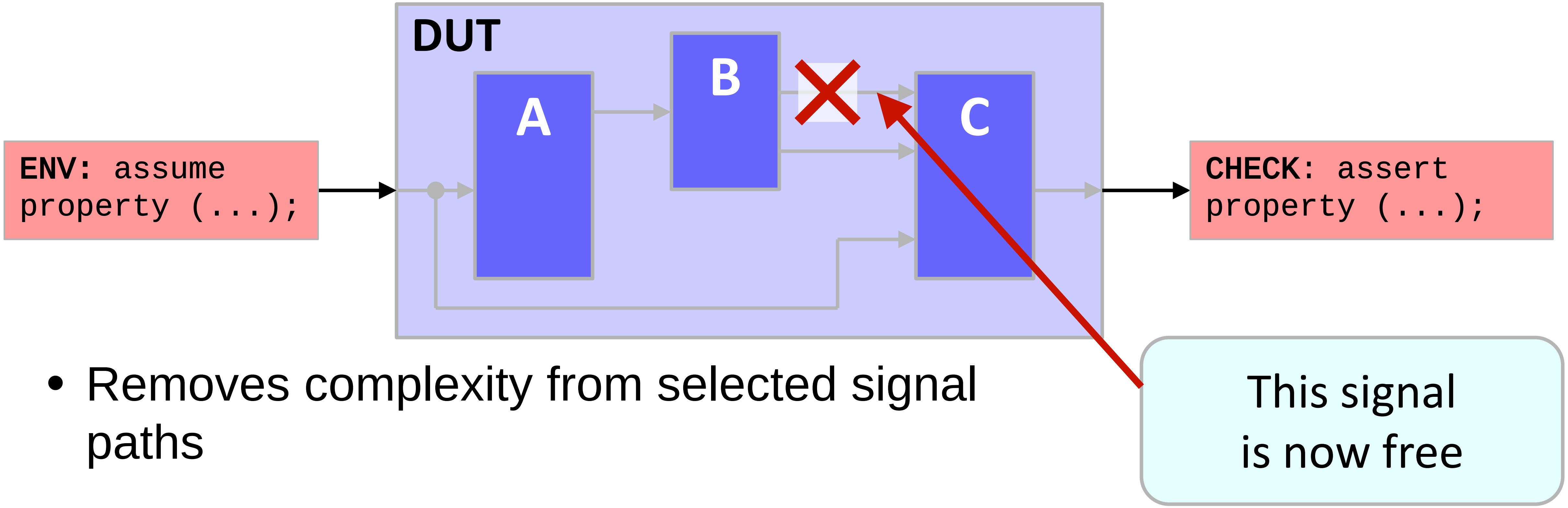
Examples:
- big math function
- long time delay

Simplify by black-boxing


DUT
A
C
ENV: assume property (...);
CHECK: assert property (...);
These signals are now free

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

Simplify using Cutpoints (stop-at)



- Removes complexity from selected signal paths

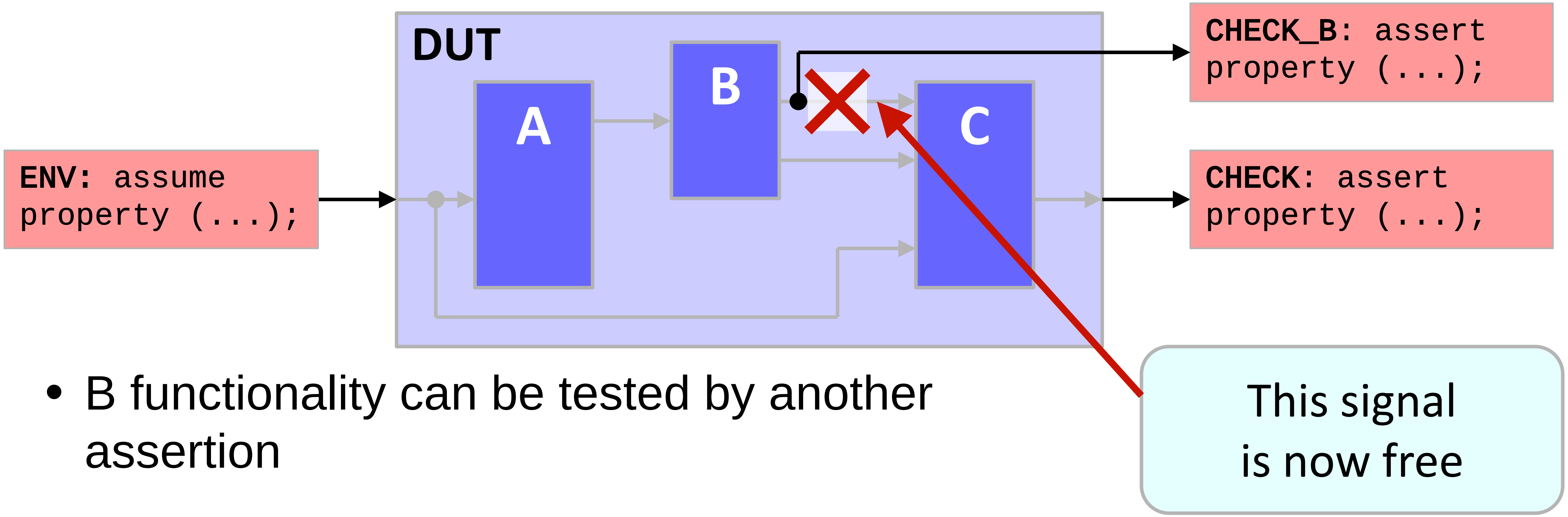
Why is it OK to destroy the design?

- 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**

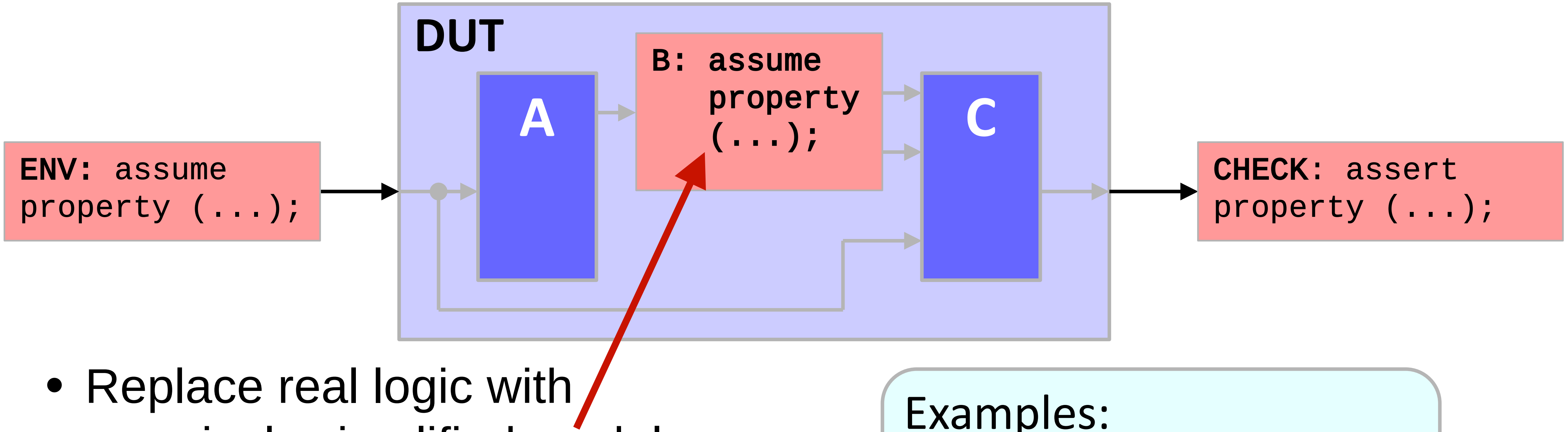
- Verification is **more** rigorous than without the cutpoint or black-box!

Verification across cutpoints


DUT
A
B
C
ENV: assume property (...);
CHECK_B: assert property (...);
CHECK: assert property (...);
This signal is now free

- B functionality can be tested by another assertion

Abstraction

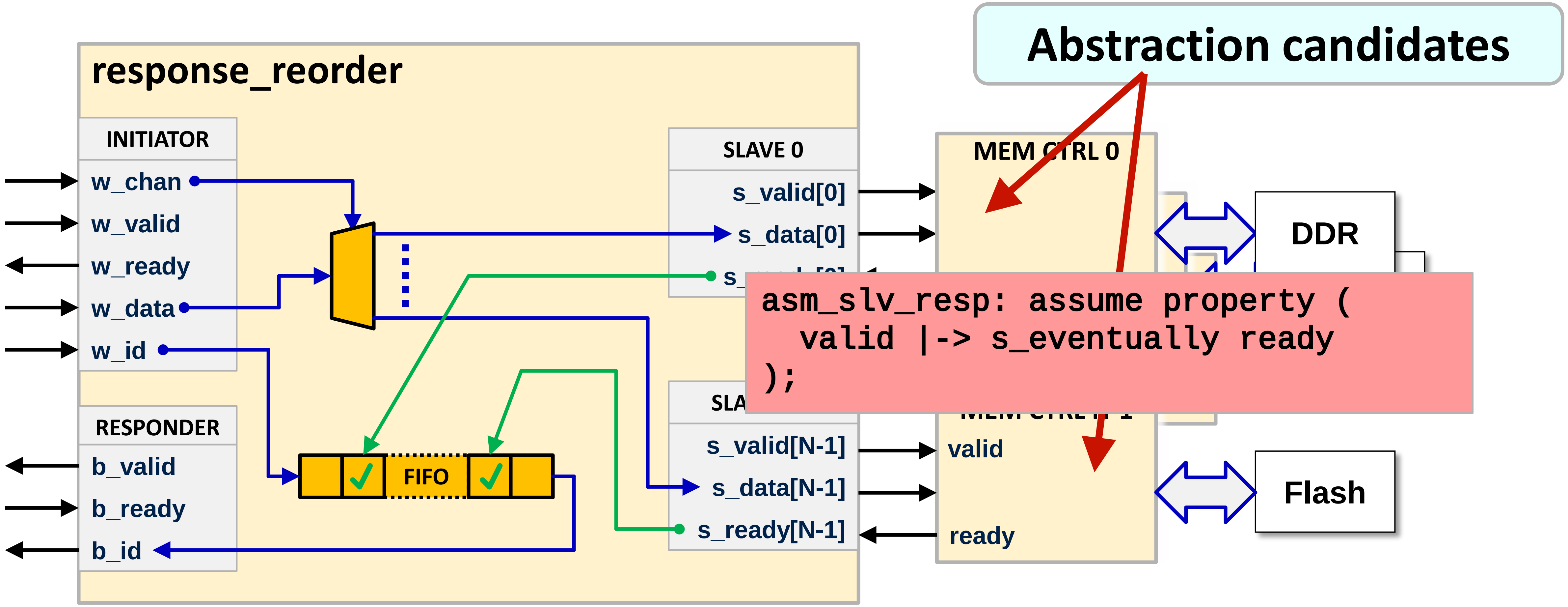

DUT
A
C
B: assume property (...);
ENV: assume property (...);
CHECK: assert property (...);

- Replace real logic with massively simplified model

Examples:
- long timeout counter
- handshake eventually

Abstraction results

- Loose similarity to AXI W-to-B channel relationship



```
asm_slv_resp: assume property (
  valid |-> s_eventually ready
);
```

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:

timeout

```
assume property (s_eventually count == LIMIT);
```

- Add custom features to avoid crazy behaviour:

other RTL may need certain specific counter behaviours

```
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

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 only as good as your spec points

Tools can provide some quality metrics

- Code coverage / reachability
- Cone of influence (COI) vs. proof coverage

"Code" coverage

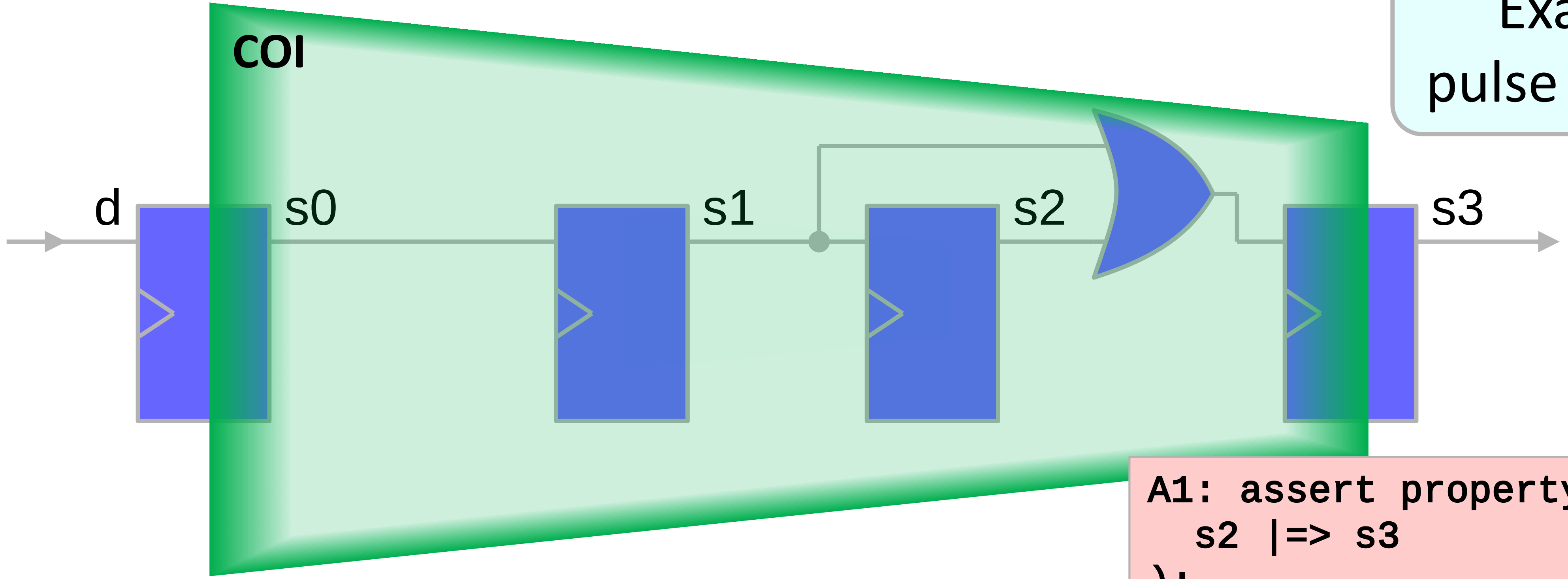
- As in simulation:
  - – code coverage as *sine qua non*
  - – helps identify weaknesses in TB
  - – can never be functionally exhaustive
  - – may need elaborate waivers

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 coverage" or similar
  - results often hard to interpret, need support from vendors

COI vs. Proof Coverage



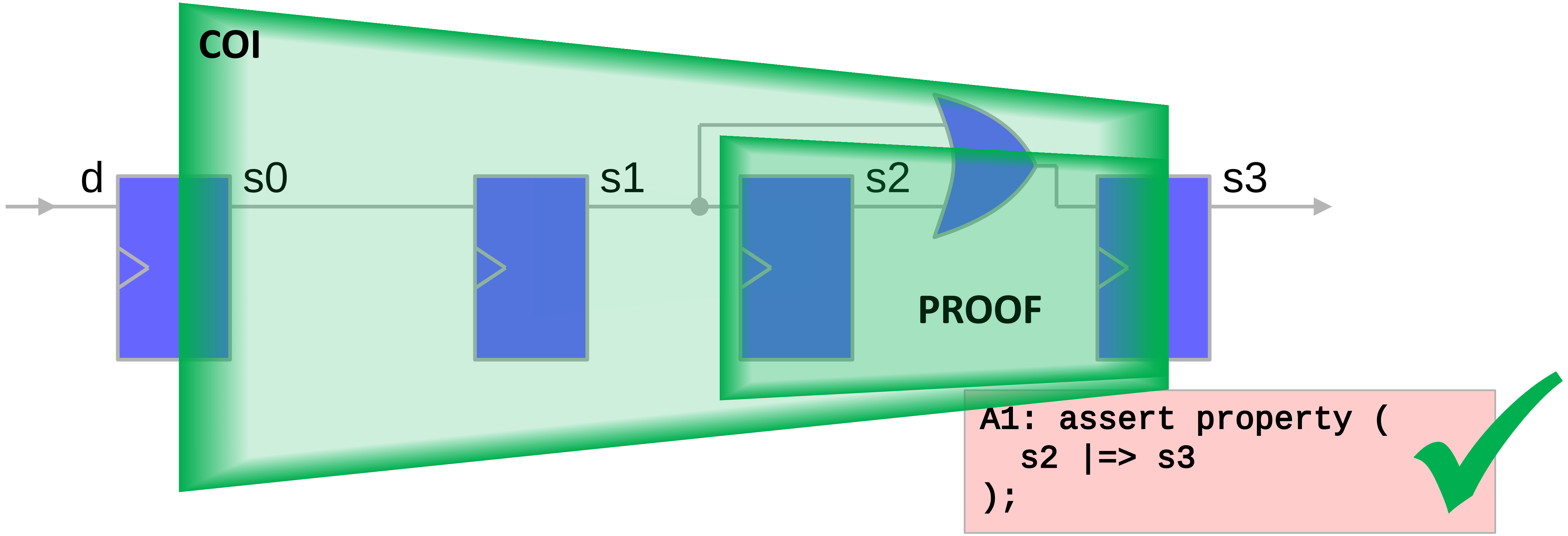
Example:
pulse stretcher

```
A1: assert property (
  s2 |=> s3
);
```

✓

- COI: All logic feeding the property's variables
  - Purely structural analysis

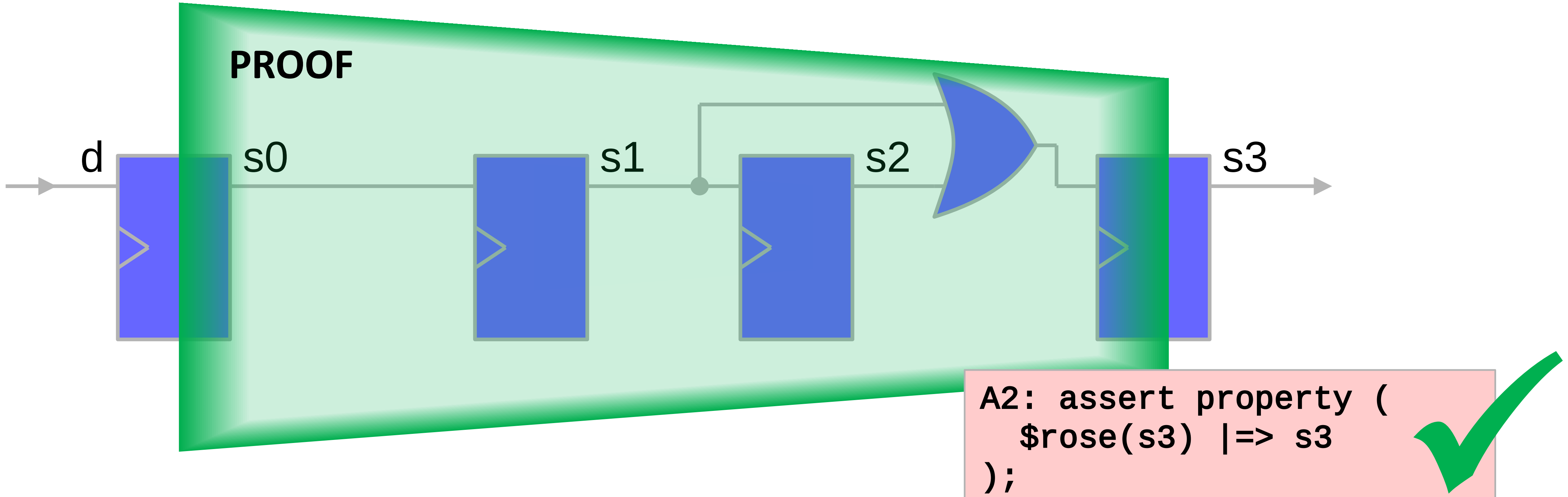
COI vs. Proof Coverage



```
A1: assert property (
  s2 |=> s3
);
```

- Proof coverage: logic needed for proof of the property
  - – Analyzes functional effect of logic

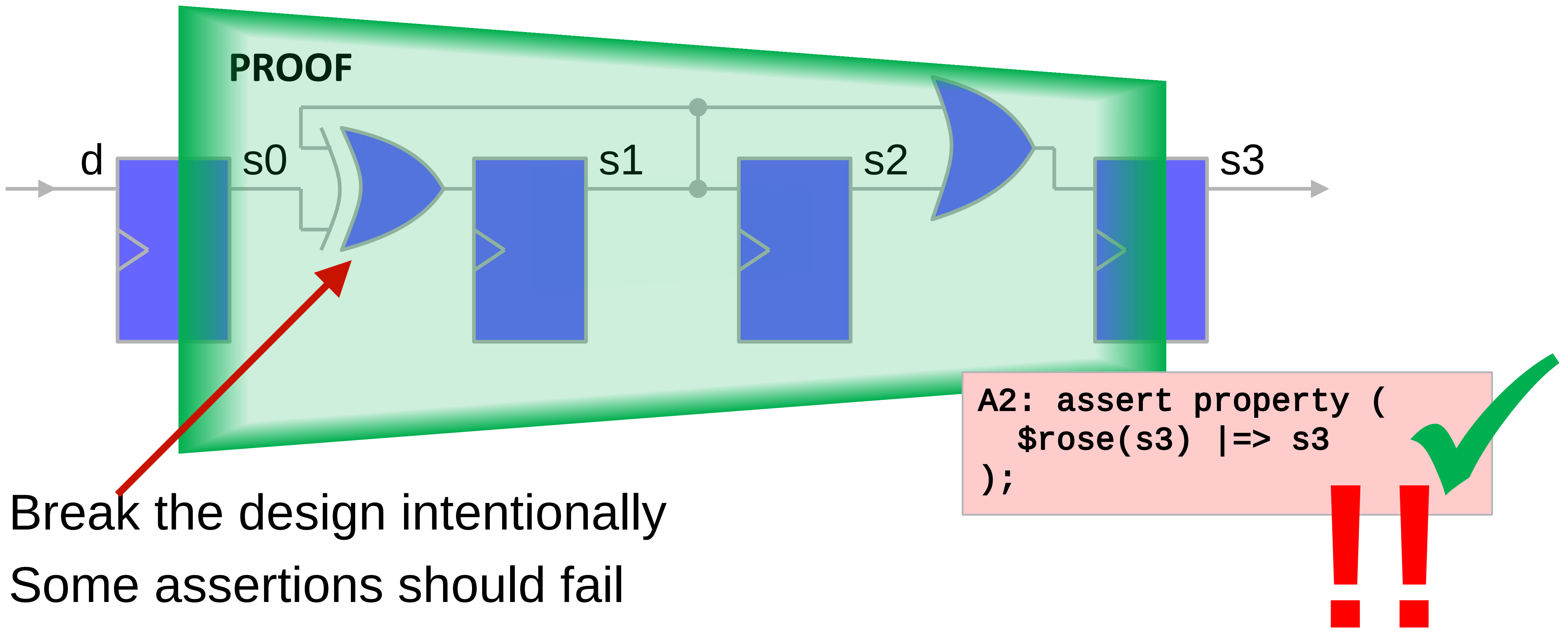
Responding to poor proof coverage



```
A2: assert property (
  $rose(s3) |=> s3
);
```

- Add assertions to get full proof coverage

Design mutation



```
A2: assert property (
  $rose(s3) |=> s3
);
```

- Break the design intentionally
- Some assertions should fail

Conclusion: Real People Still Needed

- Blind faith in tool reports is a good way to miss bugs
- No substitute for 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 and training websites
- More information related to this tutorial:

  http://www.verilab.com/resources/

  - follow "formal tutorials" link
  - many internal and external resources

Q&A