

A Formal Pot-Pourri

Laurent Ardit

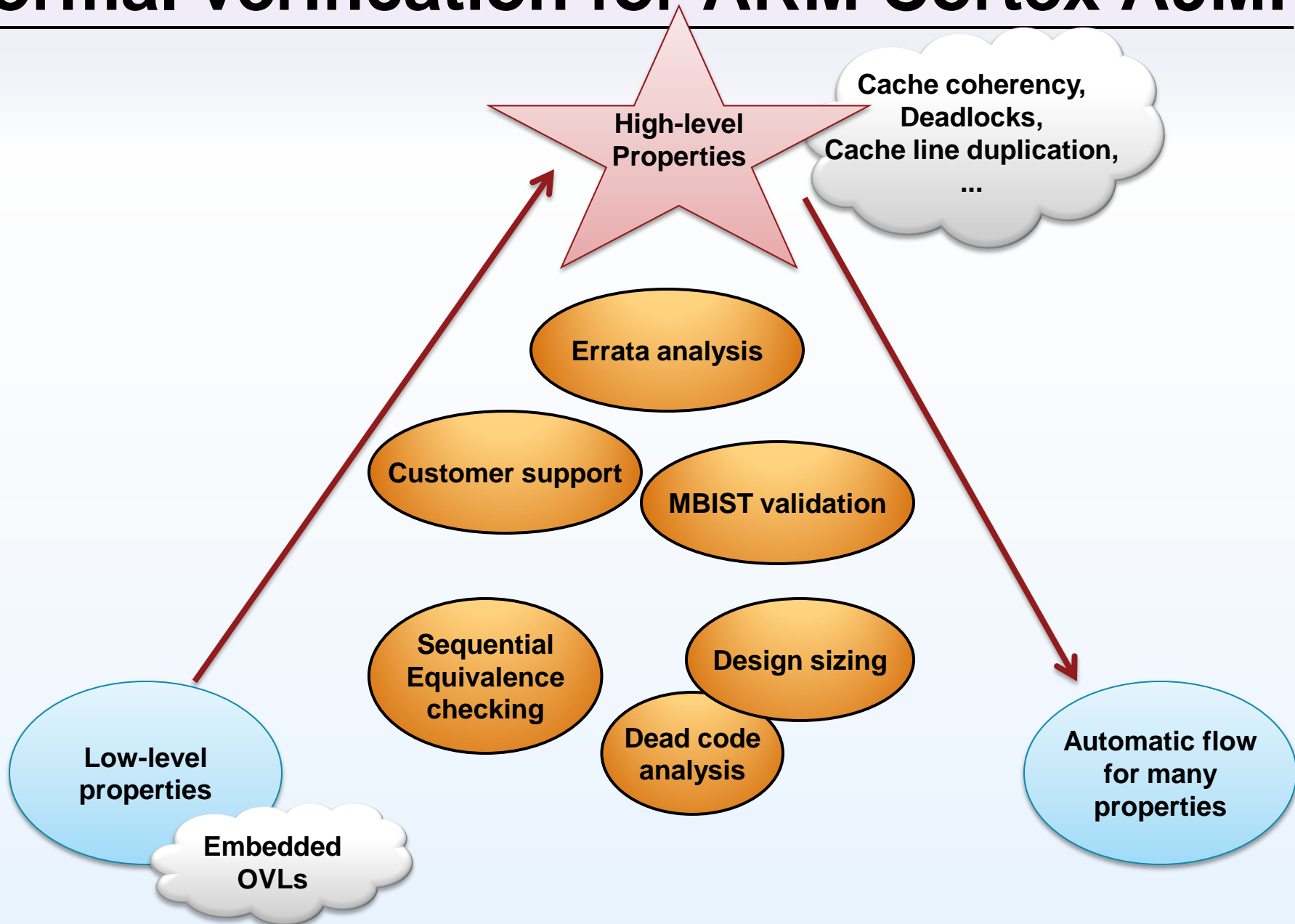
`laurent.arditi@arm.com`

ARM, Processor Division, Sophia-Antipolis, France

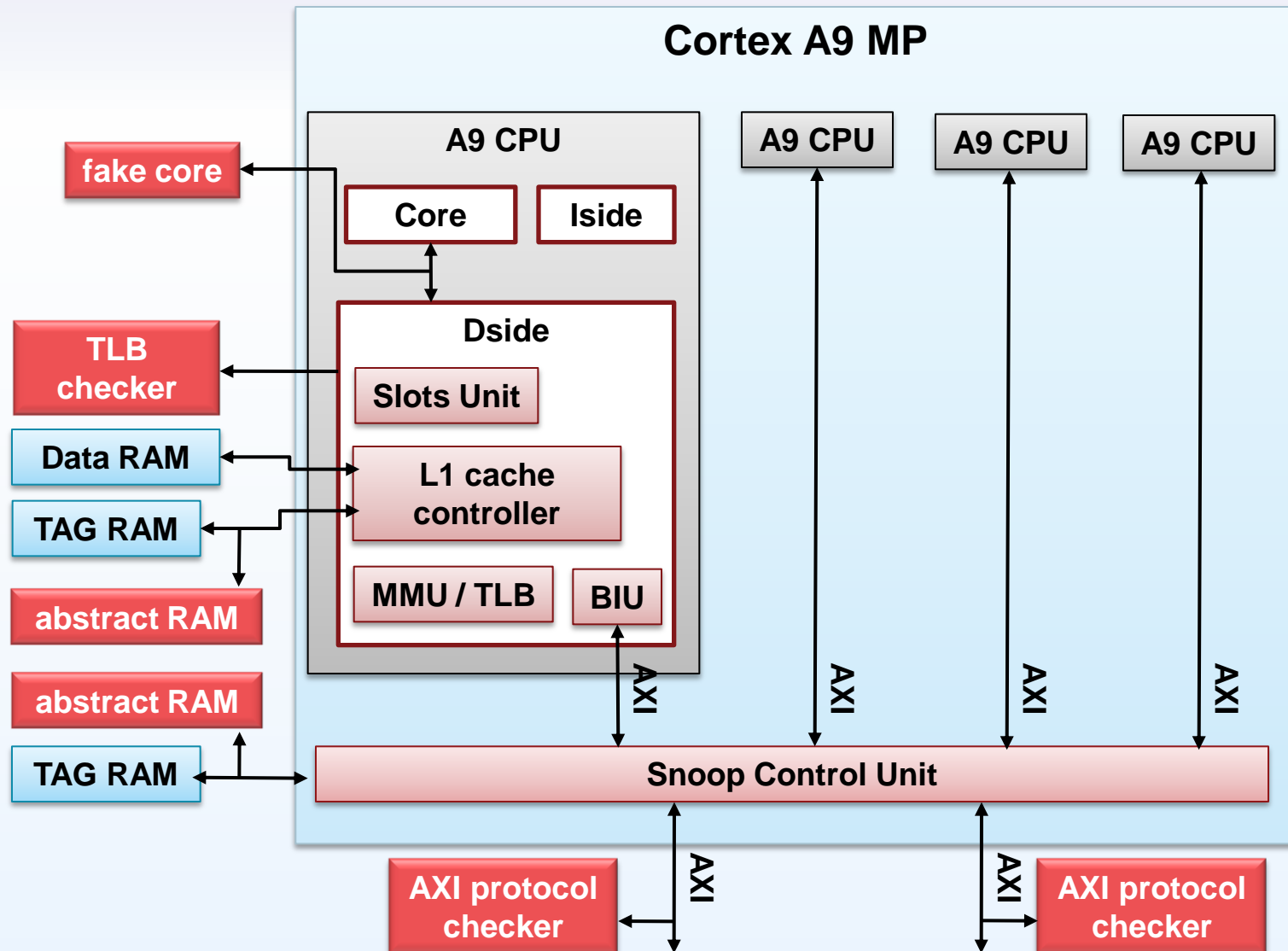
DAC User Track, June 2010



Formal verification for ARM Cortex A9MP



Cortex A9MP formal setup



High-level property verification

- High-level properties difficult to verify by simulation.
 - Cache coherency, state properties:
 - If line L is “exclusive” in one CPU, it is invalid or has a different tag in all other CPUs.
 - Cache coherency, behavioral properties:
 - When CPU₀ wants to fill line L and CPU₁ already has that line, its content moves without corruption from CPU₀ to CPU₁.
 - Absence of dead-lock / livelock
 - Lock characterization: AXI ports can always answer...eventually.
 - Cache line duplication on TAGRAM
 - If two ways have lines valid at index I, then the tags are different.

It's easy to specify properties, and to get CEXs

The challenge is to get full proofs instead of partial ones

FV for errata analysis of ARM IPs

**Bugs found
using standard
verification
flows**

**Can they be more
completely
characterized?**

**Yes, relaxing some
constraints**

**Can they be examined
by FV?**

Identify the bug as a property
Use FV tools to get a counter-
example

**All could be examined with
bug-chasing engines, within
minutes**

**In which versions
of the design are
they?**

**Usually only partial
proofs if no CEX**

**The proof
depth is
considered
as sufficient**

**Is the fix
correct?**

**Usually only
partial proofs if
no CEX**

Formal for customer and internal support

- *Customer*: “Using your IP, I observed this output trace. It is incorrect”
- *ARM*: “Could you give us details on the input sequence please?”
- *Customer*: “No!”
- *ARM –internally*: “Let’s try FV to get the inputs”

- *Customer*: “Is it possible that your IP outputs such a sequence?”
- *ARM*: “Please wait...”
- *ARM-internally*: “Guys, does anybody know if...”, “TRM doesn’t say anything about that (UNPREDICTABLE)”, “Looking at the RTL, it’s not obvious”, “Maybe, maybe not”,... , “FV will say that for sure”

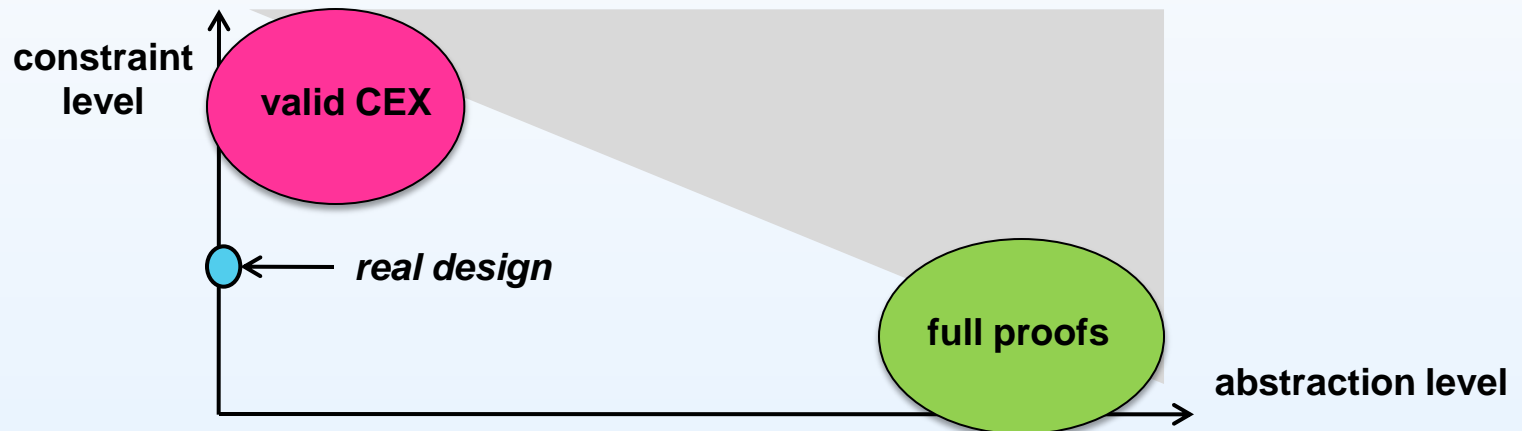
- Formal verification is also very valuable for internal “customers”:
 - dead-code detection
 - FIFO sizing
 - seq. equiv. checking
 - reset problems
 - missing clock-enables,
 - MBIST problems

Formal support stories

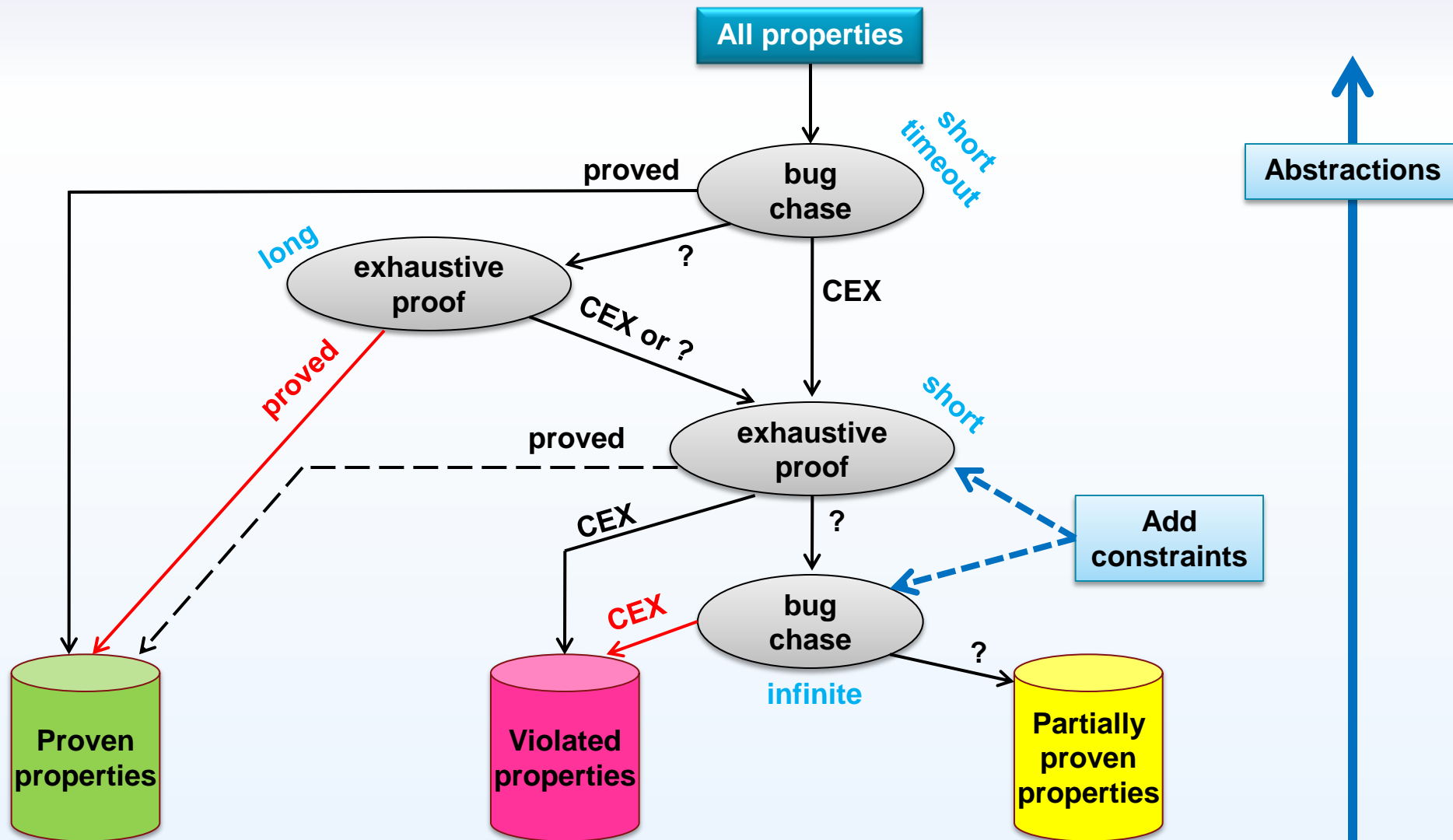
- On previous-generation L2 cache: “can these RAM control signals stay active for only 1 cycle when the latency is > 1 ?”
 - Technical leader was about to answer “No” after RTL review and simulation runs. But asked the formal guy
 - FV tool setup at the airport while waiting for a flight...
 - Result when landing in Cambridge:
 - Full proofs for all signals
 - Except for one CEX. Could be refined to identify the precise case.
 - No incorrect answer given to the customer
- On a processor core: “is the data replicated on half-word reads?”
 - Seems true looking at RTL and waveforms
 - FV identified cases where it's wrong. But they are out-of-spec.

Automatic formal flow (1)

- Hundreds of properties. Only “verified” by simulation up to product release.
- Formal flow to maximize bug detection / exhaustive proofs. But avoid having to use advanced FV techniques manually.
- Based on different abstraction and constraint levels:
 - **Abstractions are safe and incomplete**: a proof is a proof, but a CEX may be a false-negative
 - **Constraints are unsafe and complete**: a proof may be a false-positive, but a CEX is a real one



Automatic formal flow (2)



Formal flow applied to Cortex A9 assertions

- **Some bugs found.** Usually very corner cases but reachable in short instruction sequences (3 or 4, 20 to 40 cycles).
- Formal CEXs allow to identify bug nests, then targeted by directed-random simulation.
- Formal CEXs are usually easier to debug than random tests.
- Full / partial proof repartition depends on the modules:

CortexA9 blocks	Data side	Instruction side	Core
Full proofs	20%	0%	50%

- The quality of properties varies. The best ones are state encoding checks: they indirectly detect problems that are difficult to formalize:
 - local deadlocks
 - data corruptions

Formal verification tool feedback

- FV tools must be simple to use (GUI, setup, basic proof commands,...).
- But must also be configurable enough.
- Recent proof engine improvements
 - Usually excellent for bug-chasing
 - BDD engines useable only on very small designs, but for difficult proofs
 - Users need to be able to understand and select the different proof engines the tools provide.
- Features for formal agnostics are very helpful to quickly describe a behavior and to define environment constraints without property coding.
- Advanced techniques to find
 - Design abstractions
 - State abstractions
 - Invariant lemmas

are valuable but very time consuming, needing a formal engineer **and** the block designer.

General feedback on FV

- Getting exhaustive proofs is difficult compared to getting reasonable deep enough partial proofs.
- Formal is excellent to answer urgent questions safer than code review or simulation.

A good ROI in formal verification is not in trying to get full proofs, but in catching design bugs and answering design queries.