

COMS30026 Design Verification

Functional Formal Verification

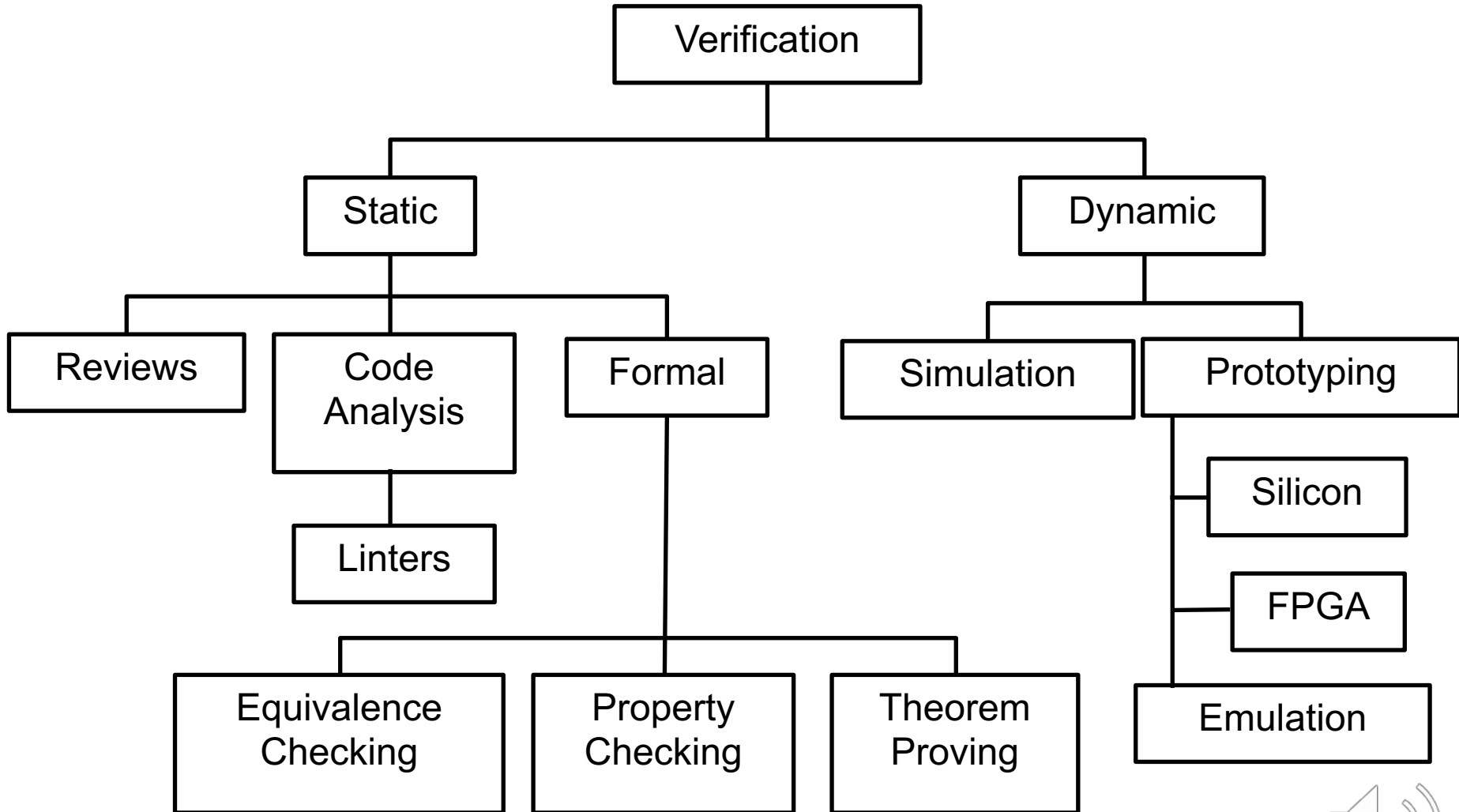
Kerstin Eder

Trustworthy Systems Laboratory

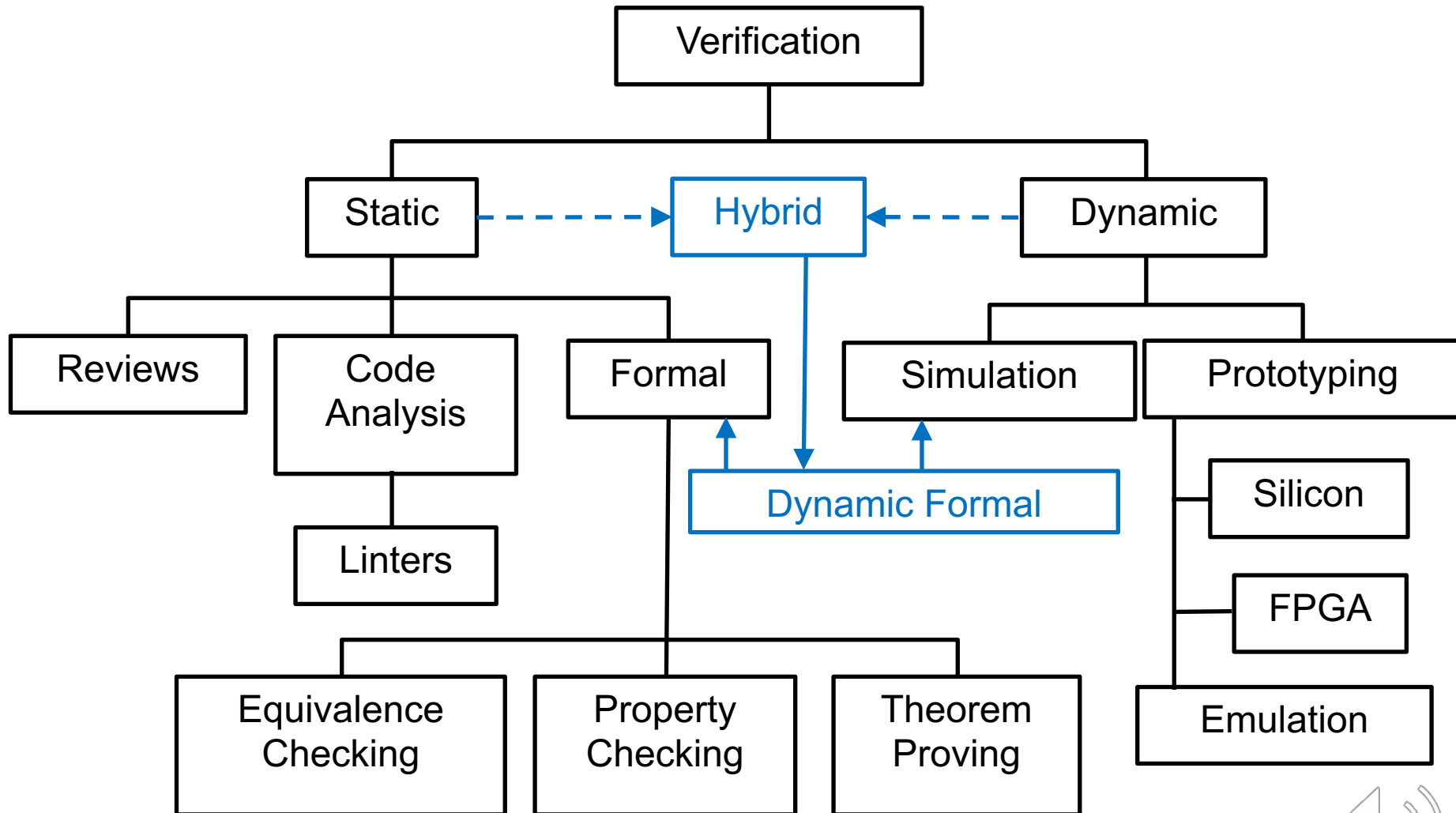
<https://www.bristol.ac.uk/engineering/research/trustworthy-systems-laboratory/>

(Acknowledgement: I gratefully acknowledge the support from Cadence who provide the licenses for the Formal Verification Tool demonstration. Special thanks also to Anton Klotz from the Cadence Academic Network.)

Functional Verification Approaches



Functional Verification Approaches



Formal Property Checking

Properties of a design (aka assertions) are formally proven or disproved.

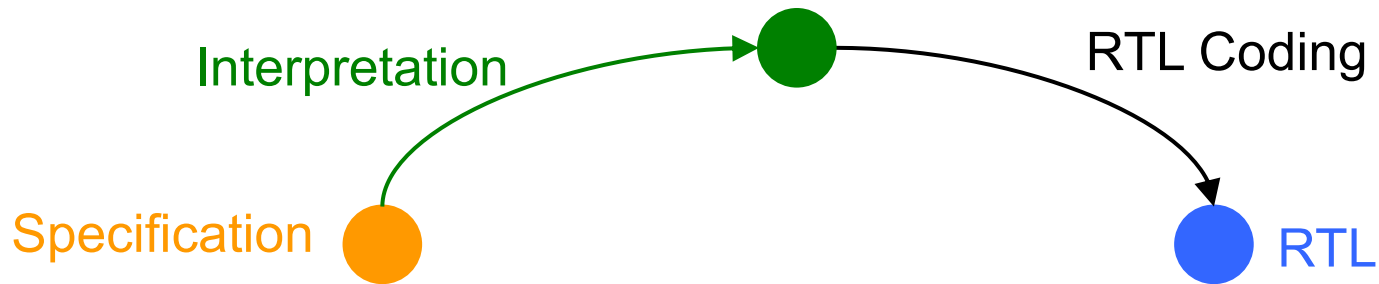
- Used to complement simulation-based verification.
- Usually employed at **lower levels** in the design hierarchy.

Give a
reconvergence model
for
formal property checking!

A **reconvergence model** is a conceptual representation of the verification process. It helps us understand **what is being verified**.

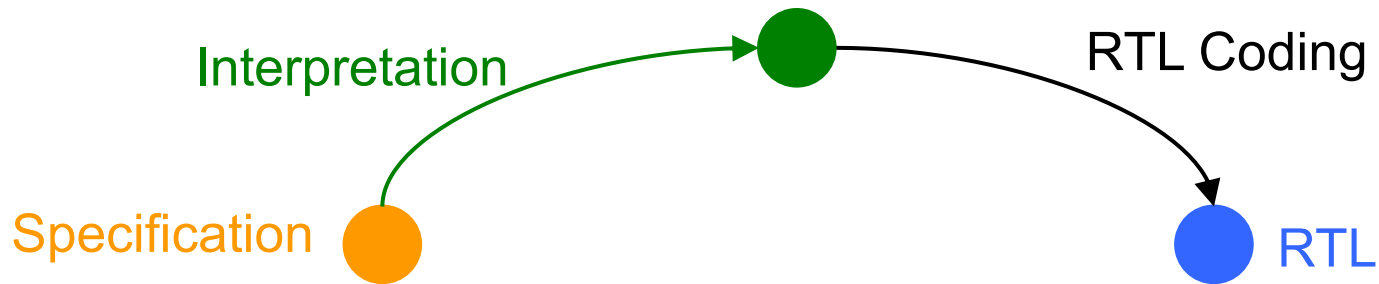


Reconvergence Model for Formal Property Checking



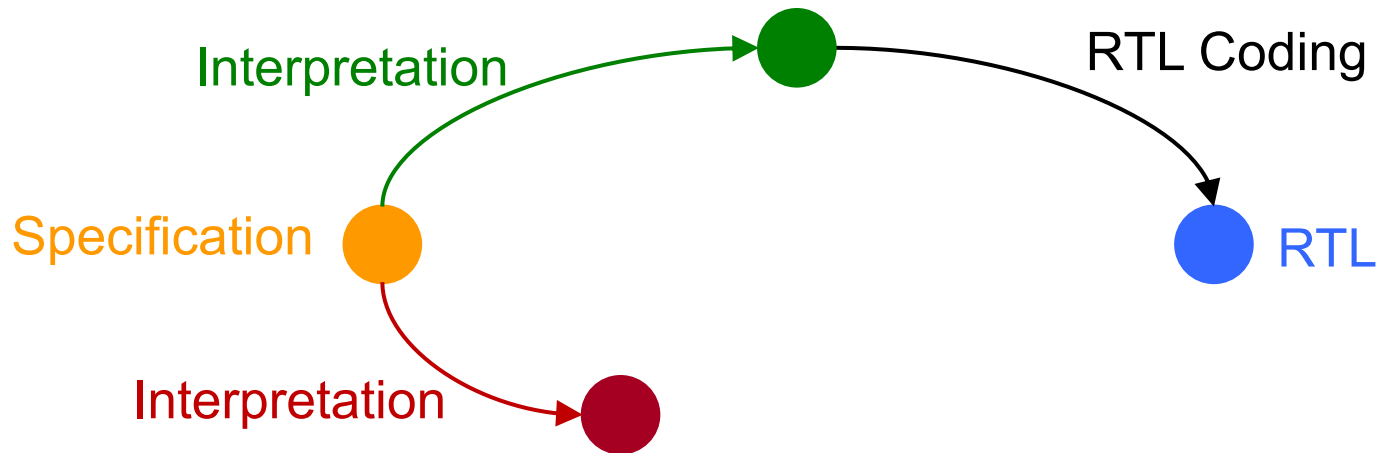
Reconvergence Model for Formal Property Checking

- **Properties** are derived from the specification. (interpretation step)
- **Properties** are expressed as formulae in some (temporal) logic.



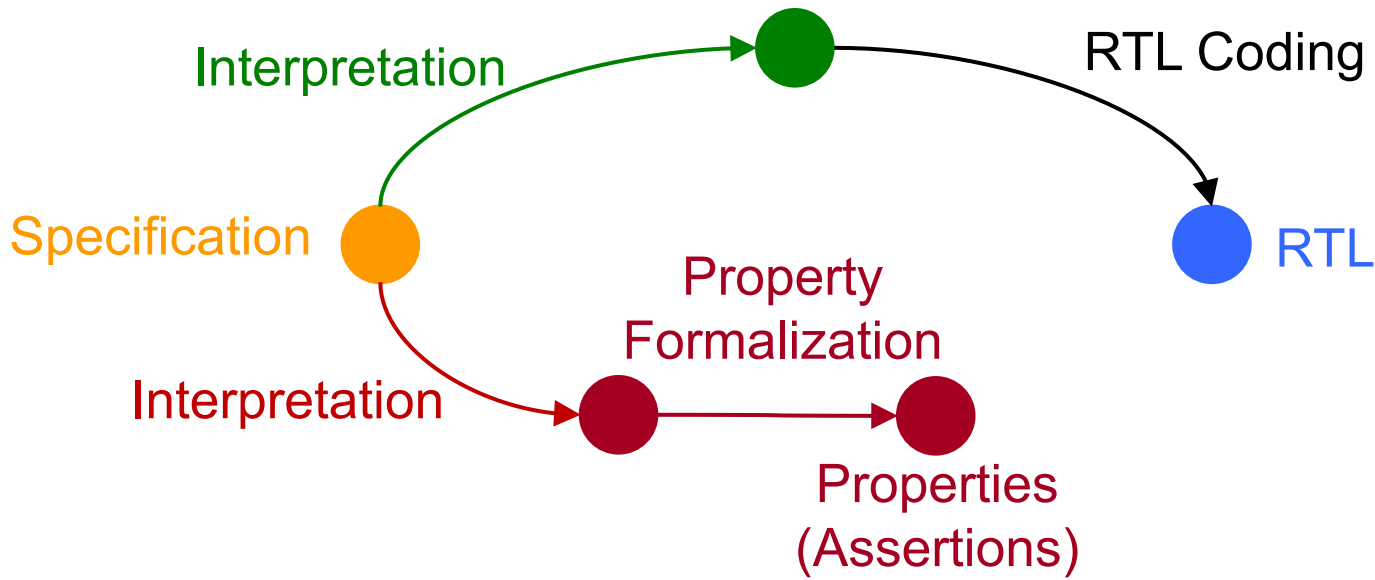
Reconvergence Model for Formal Property Checking

- **Properties** are derived from the specification. (interpretation step)
- **Properties** are expressed as formulae in some (temporal) logic.



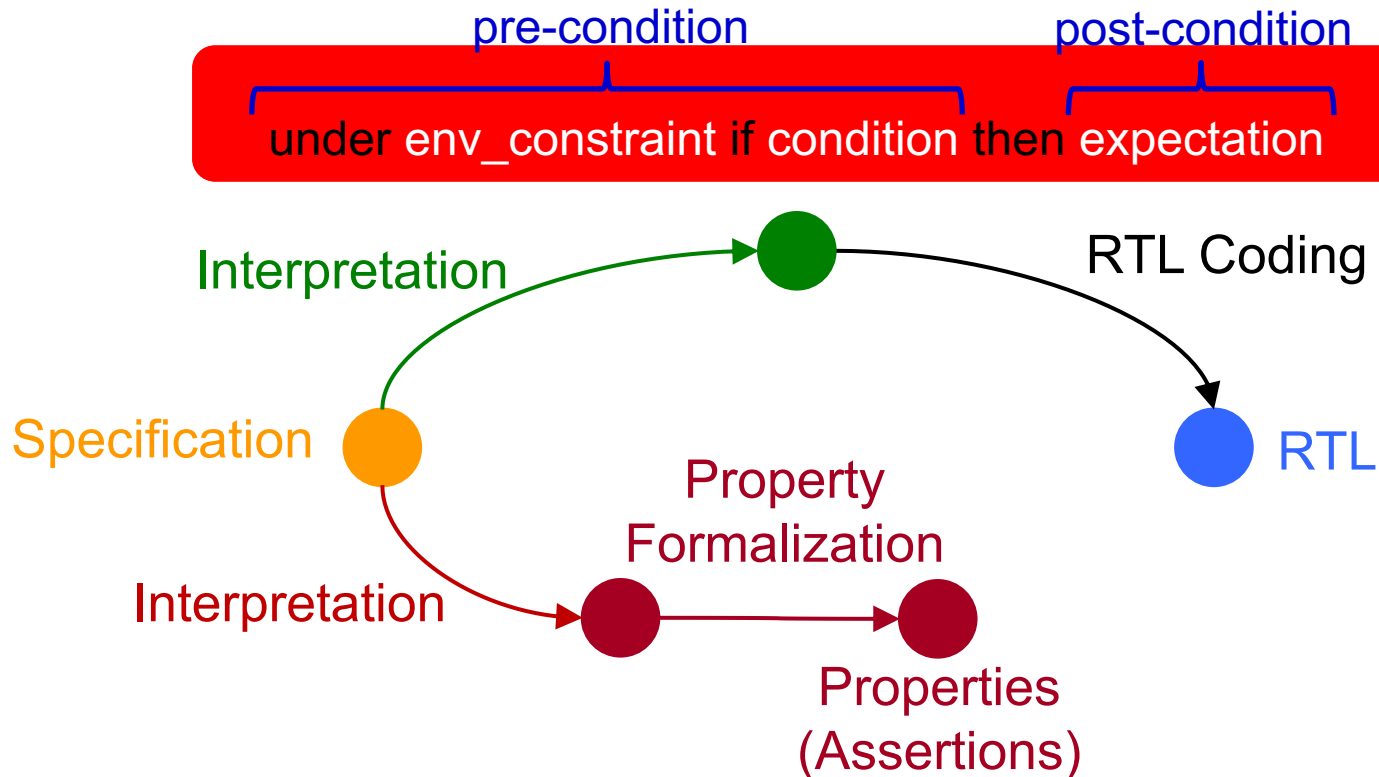
Reconvergence Model for Formal Property Checking

- **Properties** are derived from the specification. (interpretation step)
- **Properties** are expressed as formulae in some (temporal) logic.



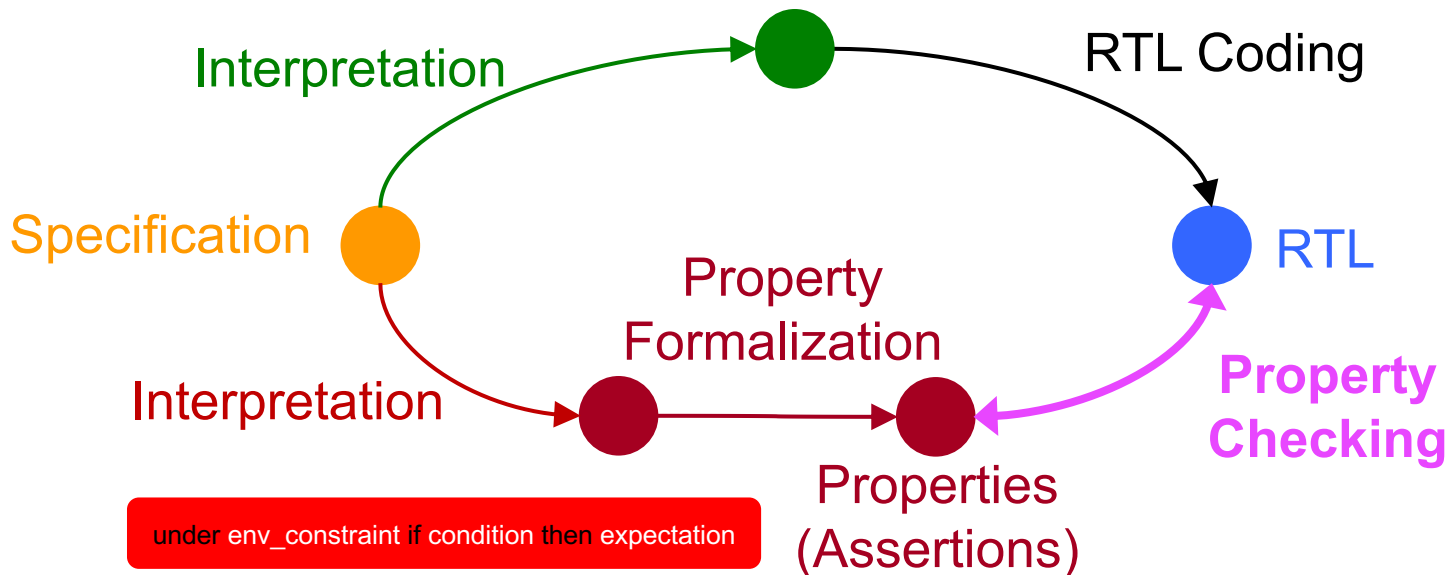
Reconvergence Model for Formal Property Checking

- **Properties** are derived from the specification. (interpretation step)
- **Properties** are expressed as formulae in some (temporal) logic.



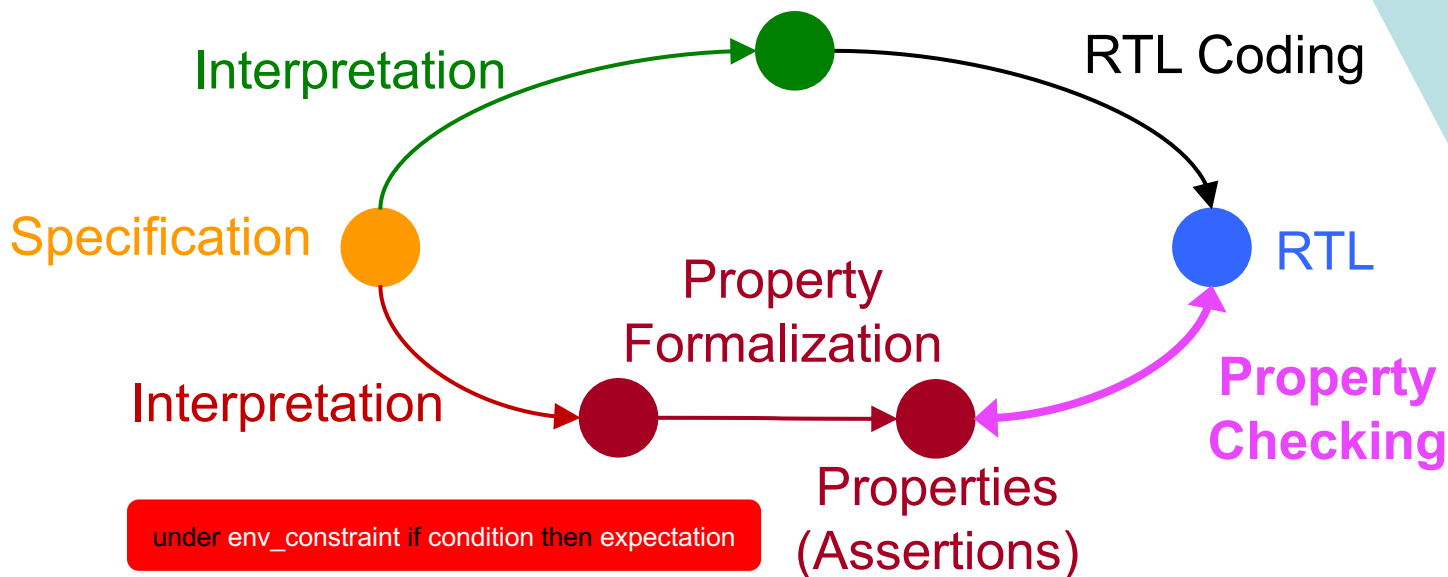
Reconvergence Model for Formal Property Checking

- **Properties** are derived from the specification. (interpretation step)
- **Properties** are expressed as formulae in some (temporal) logic.
- **Checking** is typically performed on a Finite State Machine model of the design.
 - This may be an FSM model of the **RTL** (as shown in the example).



Reconvergence Model for Formal Property Checking

- **Properties** are derived from the specification. (interpretation step)
- **Properties** are expressed as formulae in some (temporal) logic.
- **Checking** is typically performed on a Finite State Machine model of the design.
 - This may be the **RTL** (as shown in the example).



There are also Model Checkers for software, e.g. C, C++ and Java.

https://en.wikipedia.org/wiki/List_of_model_checking_tools



Overview of Formal Property Checking

- Property Checking is the **most common form** of high-level formal verification used in practice.
- Property checking is **fully automatic**.
 - Requires the properties to be written.
- It performs **exhaustive verification** of the design *wrt the specified properties*.
- It provides **proofs** and can demonstrate the **absence of bugs**.
- A **counterexample** is presented for failed properties.
- Frequently used for critical, well specified parts of the design, e.g. cache coherence protocols, bus protocols, interrupt controllers, interfaces



Scalability of Formal Verification

Due to the fact that **formal verification is exhaustive**, formal methods can suffer from **capacity limits**.

INFORMATION AND COMPUTATION 98, 142-170 (1992)

Symbolic Model Checking: 10^{20} States and Beyond*

J. R. BURCH, E. M. CLARKE, AND K. L. McMILLAN

*School of Computer Science, Carnegie Mellon University,
Pittsburgh, Pennsylvania 15213*

AND

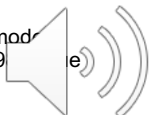
D. L. DILL AND L. J. HWANG

Stanford University, Stanford, California 94305

Many different methods have been devised for automatically verifying finite state systems by examining state-graph models of system behavior. These methods all depend on decision procedures that explicitly represent the state space using a list or a table that grows in proportion to the number of states. We describe a general method that represents the state space *symbolically* instead of explicitly. The generality of our method comes from using a dialect of the Mu-Calculus as the primary specification language. We describe a *model checking* algorithm for Mu-Calculus formulas that uses Bryant's *Binary Decision Diagrams* (Bryant, R. E., 1986, *IEEE Trans. Comput.* C-35) to represent relations and formulas. We then show how our new Mu-Calculus model checking algorithm can be used to derive efficient decision procedures for CTL model checking, satisfiability of linear-time temporal logic formulas, strong and weak observational equivalence of finite transition systems, and language containment for finite ω -automata. The fixed point computations for each decision procedure are sometimes complex, but can be concisely expressed in the Mu-Calculus. We illustrate the practicality of our approach to symbolic model checking by discussing how it can be used to verify a simple synchronous pipeline circuit. © 1992 Academic Press, Inc.

J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang. Symbolic model checking: 1020 States and beyond, Information and Computation, Volume 92, 1992, Pages 142-170, ISSN 0890-5401.

[https://doi.org/10.1016/0890-5401\(92\)90017-A](https://doi.org/10.1016/0890-5401(92)90017-A)



How big is exhaustive?

- Consider simulating a typical CPU design
 - 500k gates, 20k DFFs, 500 inputs
 - 70 billion simulation cycles,
running on 200 linux boxes for a week
 - **How big: 2^{36} cycles**



How big is exhaustive?

- Consider simulating a typical CPU design
 - 500k gates, 20k DFFs, 500 inputs
 - 70 billion sim cycles, running on 200 linux boxes for a week
 - **How big: 2^{36} cycles**
- Consider formally verifying this design
 - Input sequences: cycles $2^{(\text{inputs}+\text{state})} = 2^{20500}$
 - What about X's: 2^{15000} (5,000 X-assignments + 10,000 non-reset DFFs)
 - **How big: 2^{20500} cycles** (2^{15000} combinations of X is not significant here!)



How big is exhaustive?

- Consider simulating a typical CPU design
 - 500k gates, 20k DFFs, 500 inputs
 - 70 billion sim cycles, running on 200 linux boxes for a week
 - **How big: 2^{36} cycles**
- Consider formally verifying this design
 - Input sequences: cycles $2^{(\text{inputs}+\text{state})} = 2^{20500}$
 - What about X's: 2^{15000} (5,000 X-assignments + 10,000 non-reset DFFs)
 - **How big: 2^{20500} cycles** (2^{15000} combinations of X is not significant here!)
- These are a big numbers!
 - Cycles to simulate the 500k gate CPU design: 2^{36} (70 billion)
 - Cycles to formally verify a 32-bit adder: 2^{64} (18 billion billion)
 - Number of stars in universe: 2^{74} (10^{22})



How big is exhaustive?

- Consider simulating a typical CPU design
 - 500k gates, 20k DFFs, 500 inputs
 - 70 billion sim cycles, running on 200 linux boxes for a week
 - **How big: 2^{36} cycles**
- Consider formally verifying this design
 - Input sequences: cycles $2^{(\text{inputs}+\text{state})} = 2^{20500}$
 - What about X's: 2^{15000} (5,000 X-assignments + 10,000 non-reset DFFs)
 - **How big: 2^{20500} cycles** (2^{15000} combinations of X is not significant here!)
- These are a big numbers!
 - Cycles to simulate the 500k gate CPU design: 2^{36} (70 billion)
 - Cycles to formally verify a 32-bit adder: 2^{64} (18 billion billion)
 - Number of stars in universe: 2^{74} (10^{22})
 - Number of atoms in the universe: 2^{260} (10^{78})
 - Possible X combinations in 500k gate design: 2^{15000} ($10^{4515} \times 3$)
 - Cycles to formally verify the 500k gate design: 2^{20500} (10^{6171})

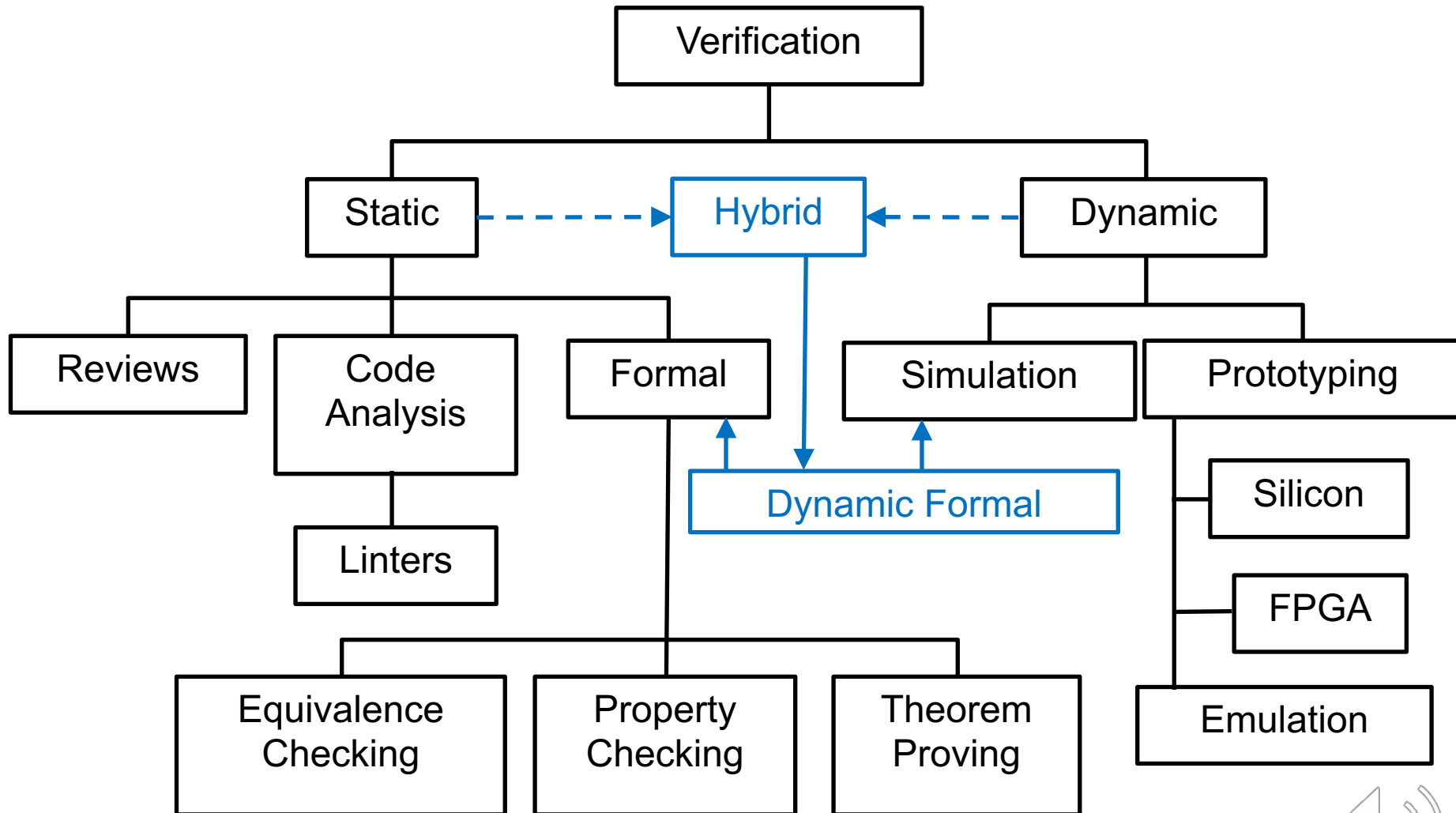


Managing complexity in FV

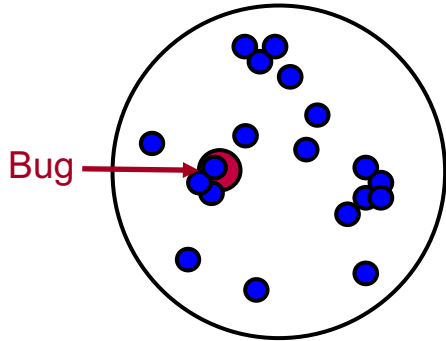
- There are tried and trusted techniques to overcome the capacity limitations of FV:
 - Start with narrow focus on block level, work up towards higher levels in the design hierarchy turning proven assertions into assumptions
 - Restrict property checking to work over finite small time windows.
 - Limit environment behaviour by strengthening constraints.
 - Case splits over a set of properties, partitioning and black boxing.



Functional Verification Approaches



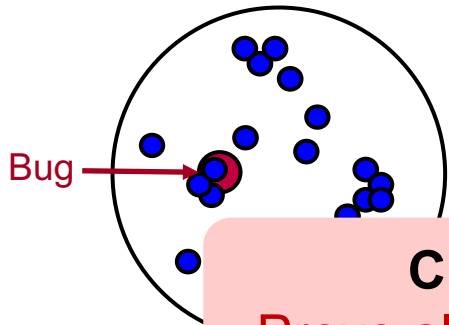
Simulation vs Formal Verification



Only selected parts
of the design can be
covered during
simulation.



Simulation vs Formal Verification



Challenge 2:
Prove all these properties.

Only selected parts
of the design can be
covered during
simulation.

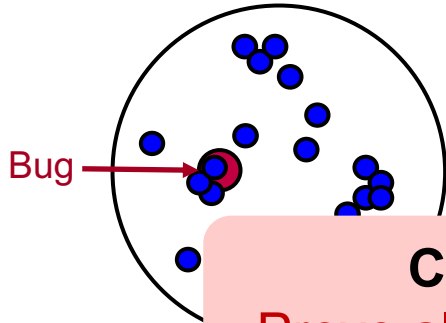
Naïve interpretation
of exhaustive formal
verification:

Verify ALL properties.

Challenge 1:
Specify
properties to
cover the entire
design.



Simulation vs Formal Verification



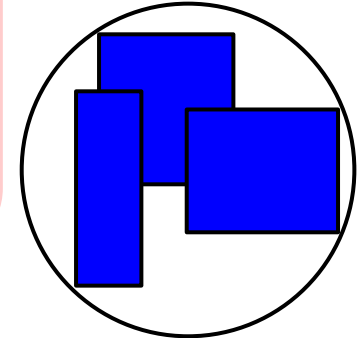
Challenge 2:
Prove all these properties.

Only selected parts of the design can be covered during simulation.

Naïve interpretation of exhaustive formal verification:

Verify ALL properties.

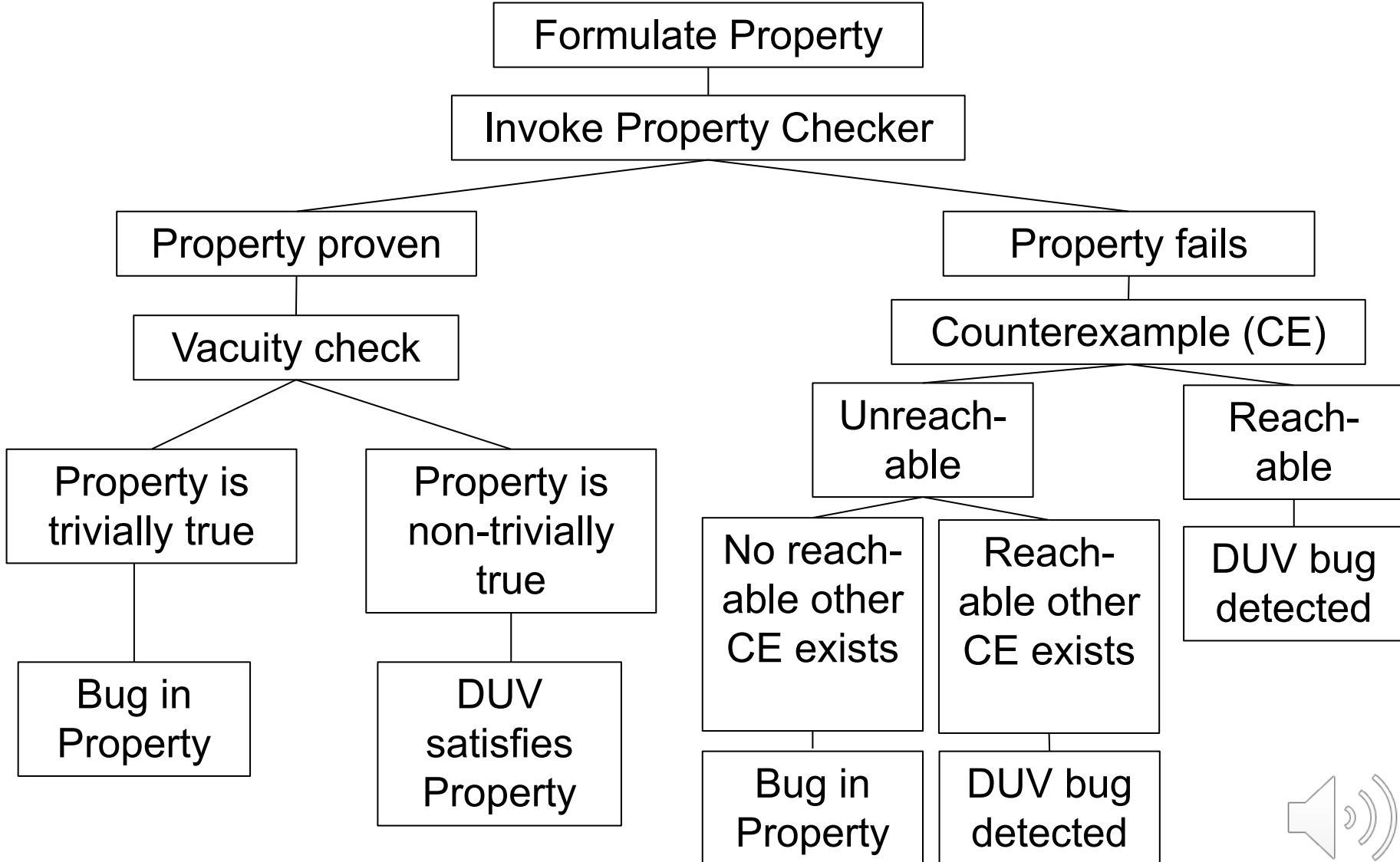
Challenge 1:
Specify properties to cover the entire design.



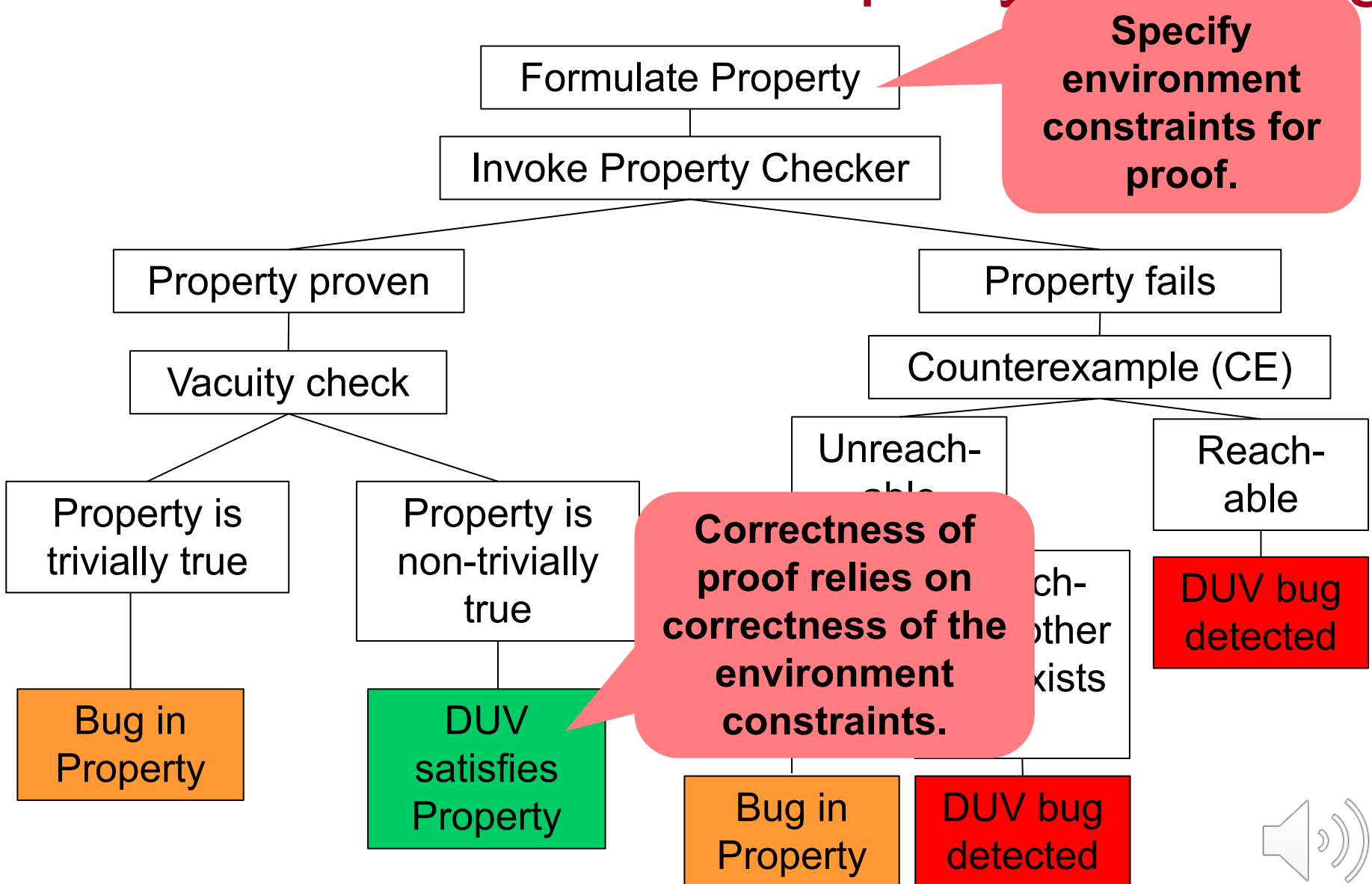
In practice, **completeness issues** and **capacity limits** restrict formal verification to selected parts of the design.



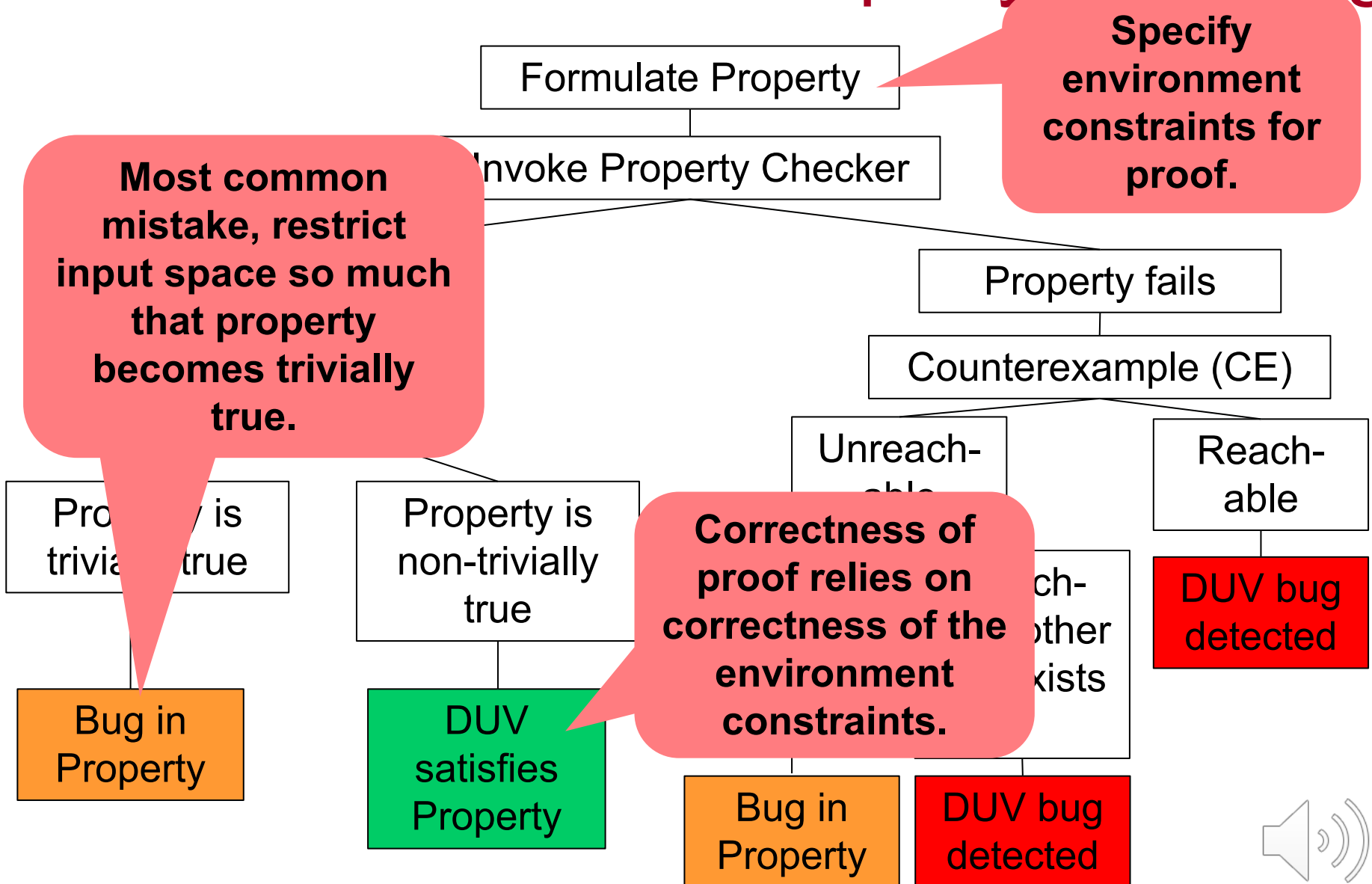
Outcomes of Formal Property Checking



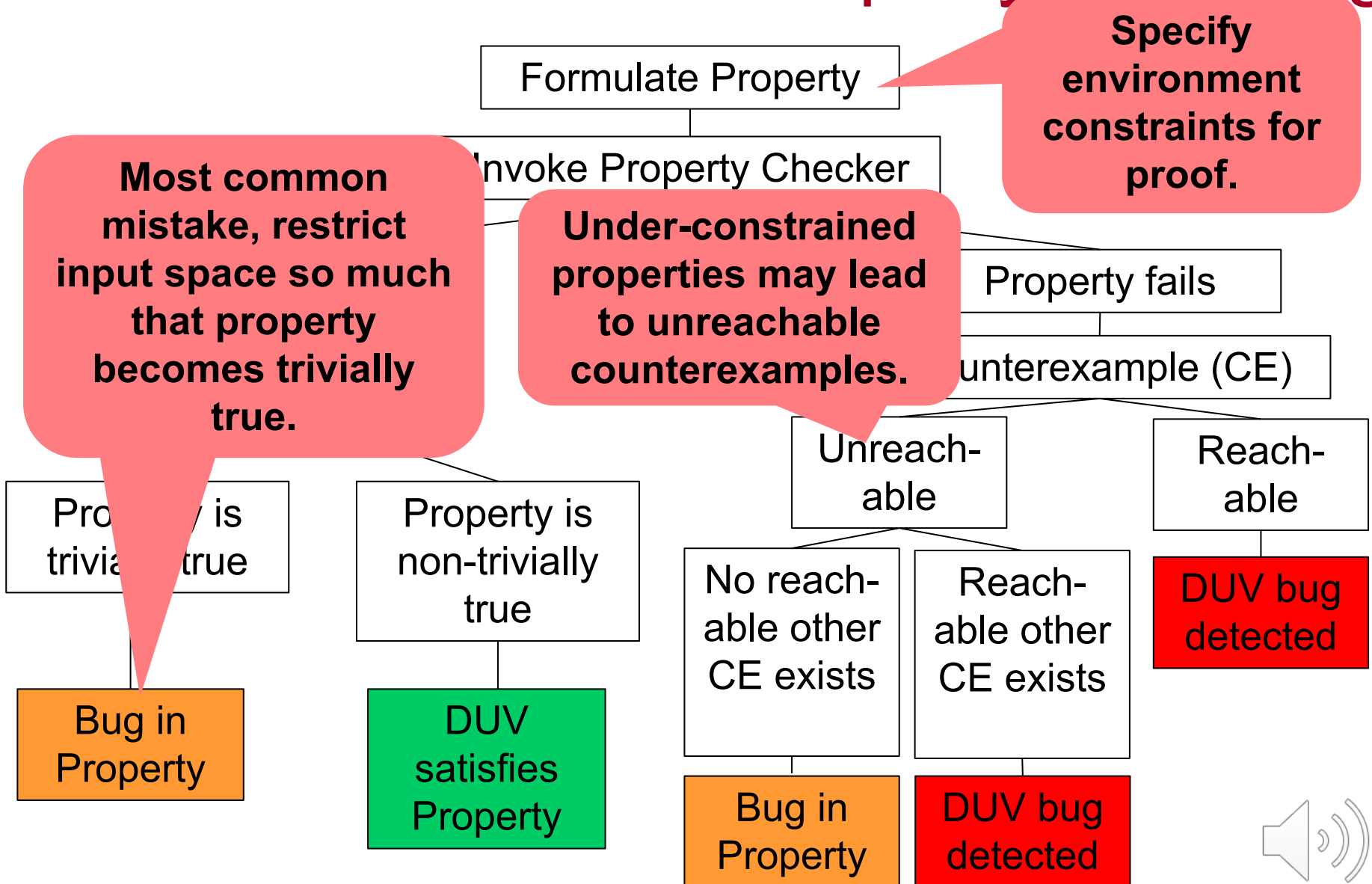
Outcomes of Formal Property Checking



Outcomes of Formal Property Checking



Outcomes of Formal Property Checking



How do you know you've encoded the property right?



- Keep properties and sequences **simple**; build complex properties from simple, short properties.
- **Peer review** properties you write.
- **Know what to expect**, e.g. create failing conditions.
- If the property fails (when you expect it to succeed), then **investigate the counterexample**:
 - Is it reachable or not?

- **But if the property succeeds, how do you know whether you've encoded the property right?**



HANDS-ON FORMAL PROPERTY CHECKING DEMO



Formal Property Checking

- Jasper DEMO

- **DUV**: FIFO design from ABV lecture
- Verification of selected FIFO properties from ABV

The screenshot displays the JasperGold Apps interface, which includes a list of formal properties and a performance graph. The interface is divided into several panes:

- Design Hierarchy:** Shows the selected component 'fifo (fifo)'.
- Property Table:** A table listing various formal properties with their verification status and performance metrics.
- Performance Graph:** A line graph showing 'Proof / Trace attempts' over 'Job Time [s]' for different engines (0.Hp, 0.Ht, 0.N, 0.B).
- Process List:** A table at the bottom showing the status of various engines and processes.

Type	Name	PRE	Hp	Ht	N	B
Assert	valid_cnt_range_top		?	15	?	14
Assert	valid_cnt_range_bottom	✓				
Assert	mutex_full_empty		✓	(-, 1)	?	4
Assert	valid_cnt_range_top		?	(1, 2)	?	4
Assert	empty_after_clear_wrong		✓	(1, 2)	?	8
Assert	empty_after_clear		✓	(1, 2)	?	8
Assert	...ty_after_clear_ignore_write		✓	(1, 2)	?	8
Assert	empty_after_clear_no_wr		✓	(1, 2)	?	8
Assert	empty_one_write_wrong		?	(1, 2)	?	2
Assert	empty_one_write		✓	(1, 2)	?	8
Assert	RW_fails		?	(1, 2)	?	4
Assert	RWEmpty		✓	(1, 2)	?	8
Assert	RWFull		✓	(1, 2)	?	8
Cover	fifo_not_full		✓	(1, 1)	?	1
Cover	fifo_full		?	(1, 5)	?	11
Cover	fifo_empty		✓	(1, 1)	?	1
Cover	fifo_not_empty		✓	(1, 1)	?	1

Index	Engine	PID	Host	Status	Time	Memory (Resident)	Progress	Proof
0	PRE				0.0 s			
0	Hp		snowy.cs.bris.ac.uk	Stopped	0.1 s	10.99 MiB		
0	Ht		snowy.cs.bris.ac.uk	Stopped	0.1 s	9.33 MiB		
0	N		snowy.cs.bris.ac.uk	Stopped	0.1 s	8.97 MiB		
0	B		snowy.cs.bris.ac.uk	Stopped	0.1 s	7.19 MiB		



The demo session includes

- **Automatic generation of basic properties using “Visualize”:**
 - Basic functionality of the DUV
 - Range checks of signals
- **Verification of SVA properties:**
 - “Empty and full are never asserted together.”
 - “After clear the FIFO is empty.”
 - “On empty after one write the FIFO is no longer empty.”
- **Inspect and understand counterexamples:**
 - Debug several failed properties

- Note:**
- Close link to coverage closure (by construction).
 - Link from **env_constraints** to simulation assertions.



Summary

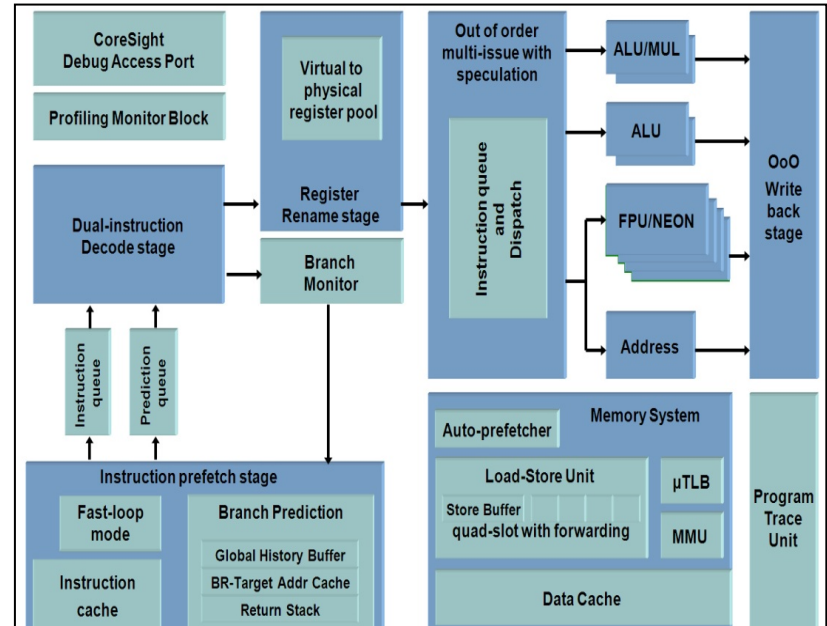
Functional Formal Verification

- Distinction between static and dynamic verification techniques
- Reconvergence model for formal verification
- What happens during formal verification
- Capacity limits and techniques to manage complexity
- Simulation vs. formal verification
- Outcomes of formal property checking
- Guidelines on writing properties



Conclusion

No single method is adequate to verify a whole design in practice.



- Carefully select the verification methods that maximize ROI for each level in the design hierarchy.
- Complement simulation with formal verification techniques to exploit the benefits and mitigate the limitations of each technique.



