

Formal Verification Bootcamp

Mike Bartley
CEO and Founder
Test and Verification Solutions Ltd

February 2019



Practical Issues

- Refreshments
- Mobile Phones
- Fire
- Acknowledgements
 - To Srikanth Vijayaraghavan for allowing us to use examples from “A Practical Guide for System Verilog Assertions”
 - To Alexandre Esselin Botelho of Cadence for help in preparing the course

Objectives

- The tutorial is not about
 - Learning SVA
 - Although we try to cover enough to be able to write assertions
 - Becoming FV experts
 - For example, how to use cut points, complex models and abstractions, ...
 - Learning a particular tool
 - The tools are used as a vehicle to give some experience in writing and proving properties
 - You need to contact your tool vendor to get an evaluation, license, training, etc.

Objectives

- The tutorial is about
 - Using some SVA
 - properties, covers, assumptions
 - Some basic FV experience
 - To gain an appreciation
 - Understanding how best to incorporate formal into your design flows and your organisation
 - Formal verification adoption has many potential hazards

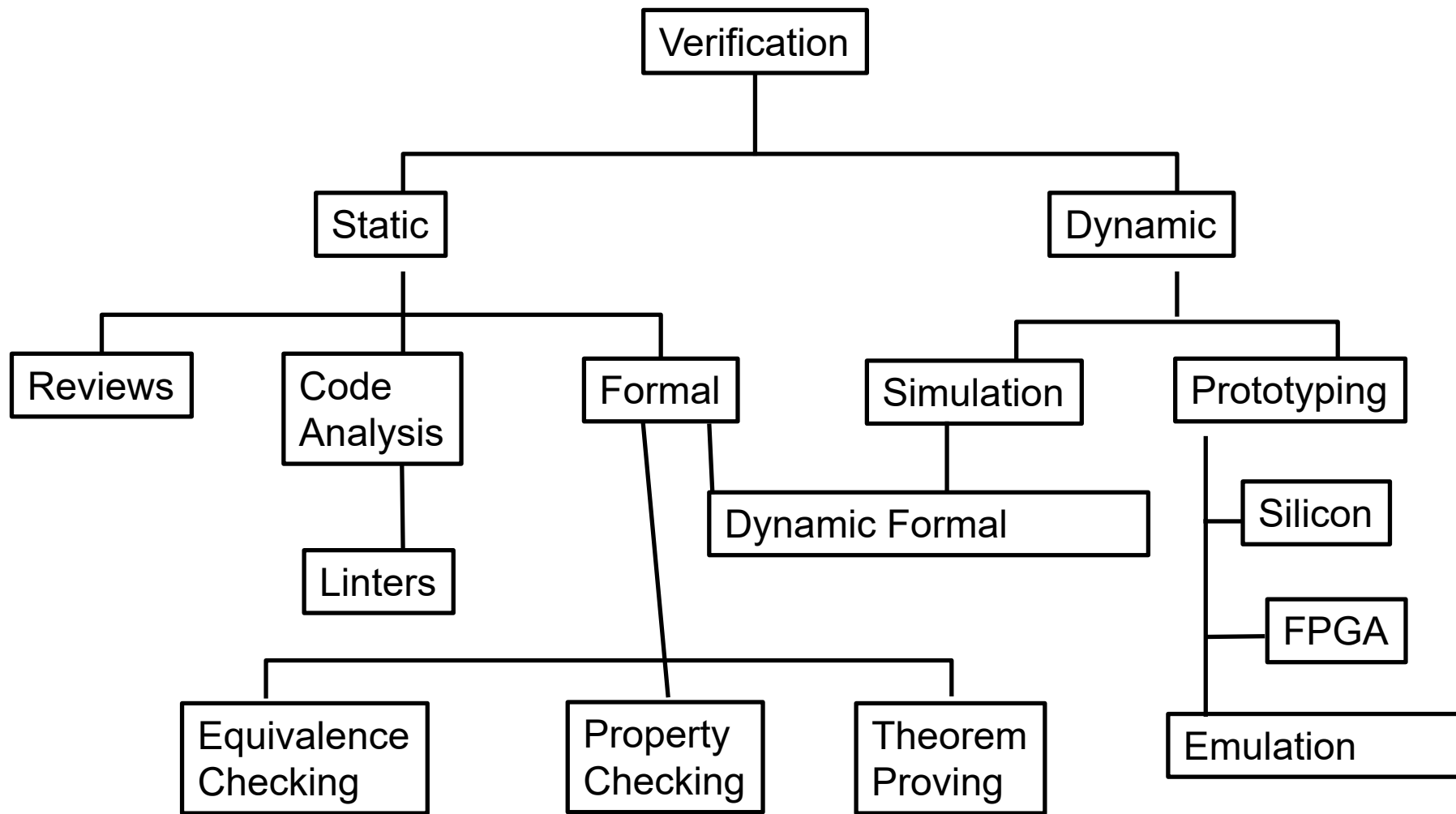
Your speaker: Mike Bartley

- PhD in Mathematical Logic
- MSc in Software Engineering
- MBA
- Worked in software testing and hardware verification for over 25 years
 - ST-Micro, Infineon, Panasonic, ARM, NXP, nVidia, ClearSpeed, Gnodal, DisplayLink, Dialog, ...
 - Worked in formal verification of both software and hardware
- Started T&VS in 2008
 - Software testing and hardware verification products and services
 - Offices in UK, India, USA, Singapore, Japan and Germany

Introduction

Quick Overview of Property Checking

Functional Verification Approaches



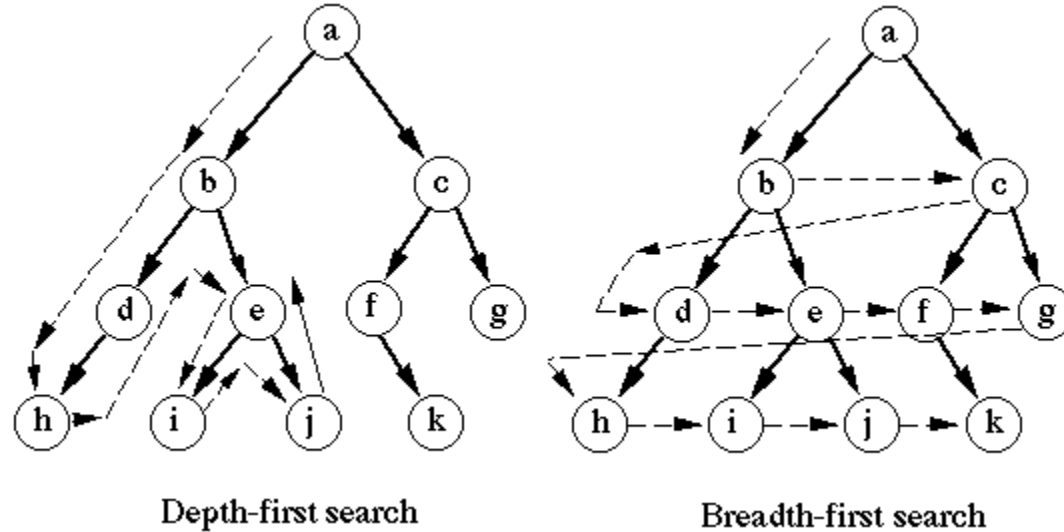
Introduction: Role of Simulation

- Most widely used verification technique in practice
- Complexity of designs makes **exhaustive simulation impossible** in terms of cost/time.
 - Engineers need to be selective
 - Employ state of the art coverage-driven verification methods
 - Test generation challenge
- Simulation can drive a design deep into its state space.
 - Can find bugs buried deep inside the logic of the design
- Understand the limits of simulation:
 - **Simulation can only show the presence of bugs but can never prove their absence!**

Introduction: Formal Property Checking



- Define properties of a design with the following aim
 - To formally prove
 - Or disprove and find a bug
- Typical flow
 - Properties are derived from the specification.
 - Properties are expressed as formulae in some (temporal) logic.
 - Checking is typically performed on a model of the design.
 - Usually the RTL
- Traditionally employed at **higher levels** of abstractions
 - But tool capacity
 - And assertion-based verification
 - Has widened their application

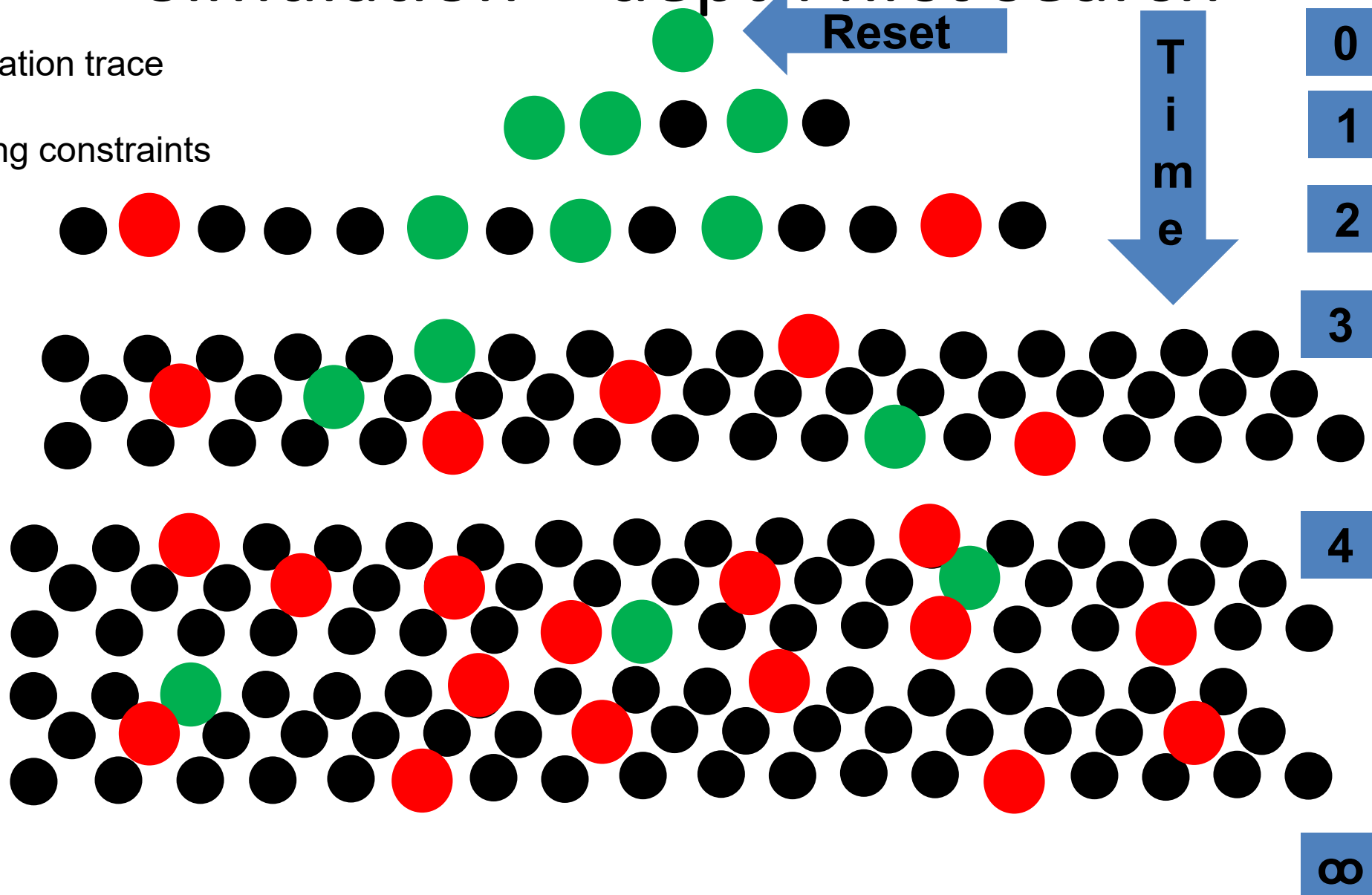
Simulation Depth-first vs. Formal Breadth-first



- Where the nodes are states in the simulation
- And the arcs are clocked transitions
- **But the trees are**
 - Very wide
 - Very deep

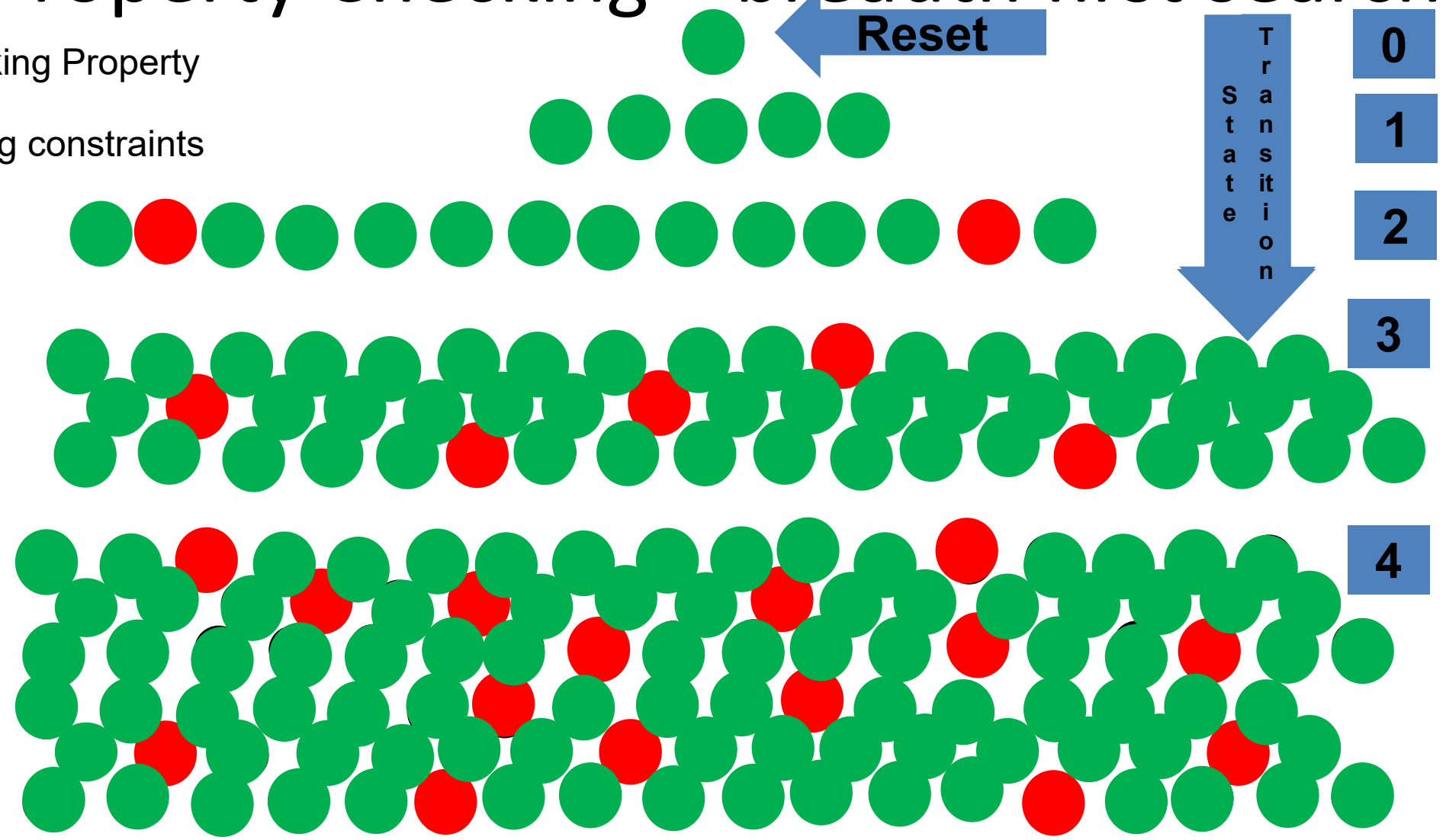
Simulation – depth first search

-  Simulation trace
-  Adding constraints



Property Checking – breadth first search

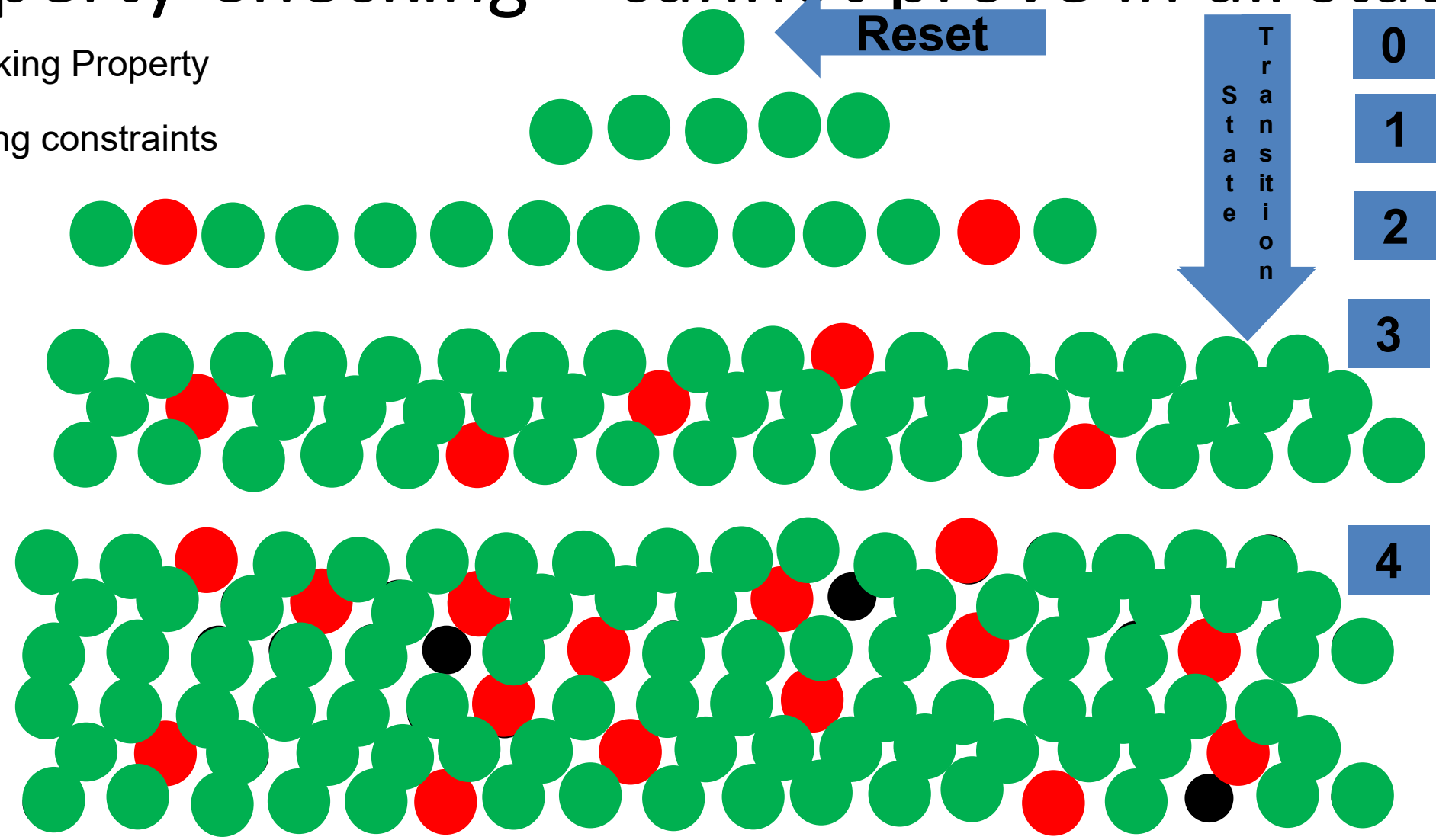
- Checking Property
- Adding constraints



Finite State Space

Property Checking – cannot prove in all states?

- Checking Property
- Adding constraints



Introduction to Simulation and Verification

Challenge 1:
Specify properties to
cover the entire design.

Challenge 3:
Proving you have
covered the design.

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.

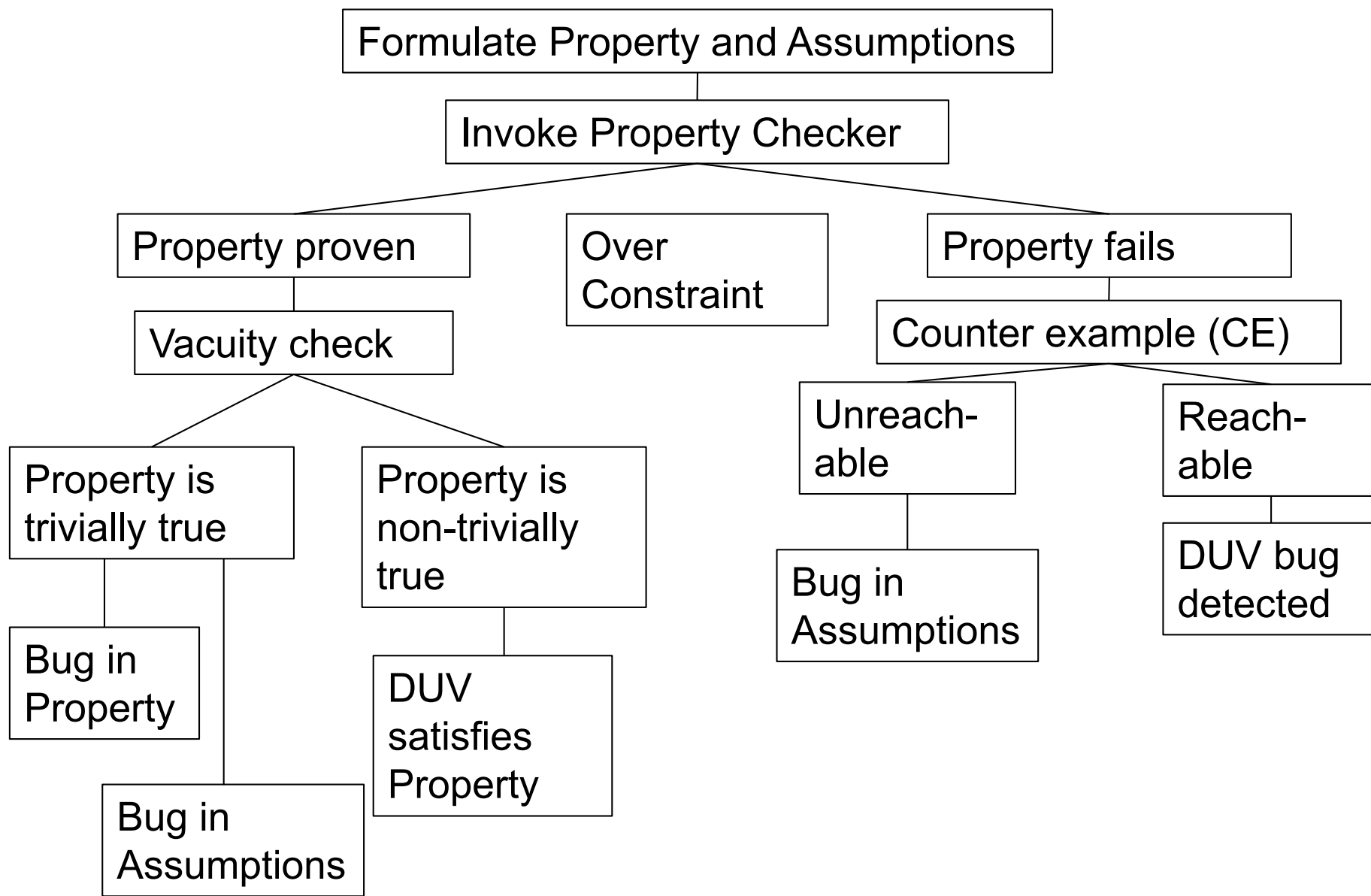
In practice, **completeness
issues** and **capacity limits**
restrict formal verification to
selected parts of the design.

Property Checking – a very brief introduction

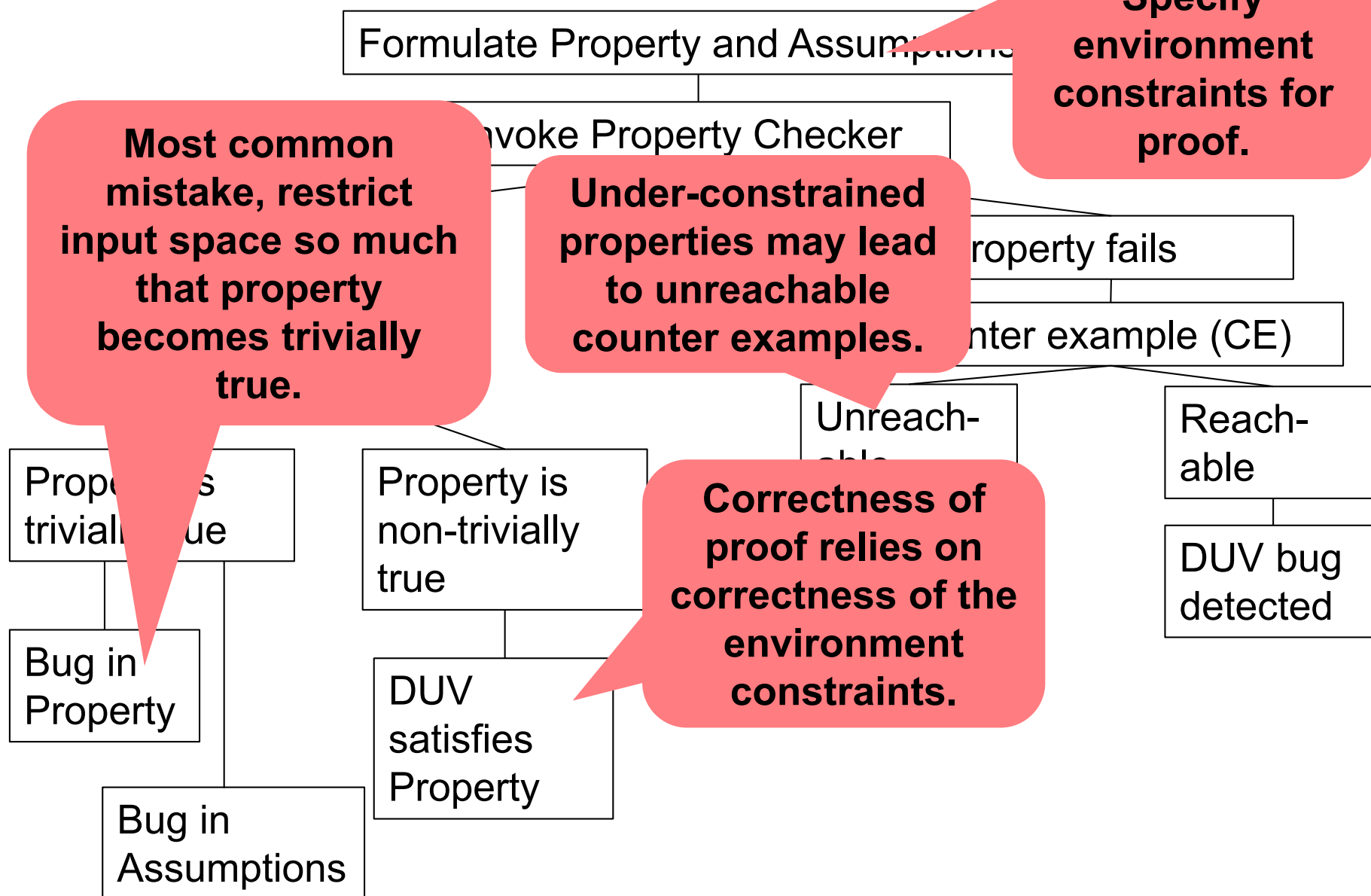
Inputs to the tool

- | | |
|---|--|
| <ul style="list-style-type: none">• 3 inputs to the tool<ul style="list-style-type: none">– A model of the design– A property or set of properties representing the requirements– A set of assumptions, expressed in the same language as the properties<ul style="list-style-type: none">• typically constraints on the inputs to the design | <ul style="list-style-type: none">• For example<ul style="list-style-type: none">– Usually RTL– Items are transmitted to one of three destinations within 2 cycles of being accepted<ul style="list-style-type: none">• $(req_in \ \&\& \ gnt_in) \rightarrow \#\#[1:2]$
$(rec_a \ \ rec_b \ \ rec_c)$– The request signal is stable until it is granted<ul style="list-style-type: none">• $(req_in \ \&\& \ !gnt_out) \rightarrow \#\#1 \ req_in$• We would of course need a complete set of constraints |
|---|--|

Outcomes of Formal Property Checking



Outcomes of Formal Property Checking



Assertion-Based Verification

Types of Assertions: Safety Properties

- **Safety:** Something bad does not happen
 - The FIFO **does not** overflow.
 - The system **does not** allow more than one process to use a shared device simultaneously.
 - Requests are answered within 5 cycles.
- More formally: *A safety property is a property for which any path violating the property has a finite prefix such that every extension of the prefix violates the property.*

[Accellera PSL-1.1 2004]

Safety properties can be falsified by a finite simulation run.

Types of Assertions: Liveness Properties

- **Liveness:** Something good eventually happens
 - The system **eventually** terminates.
 - Every request is **eventually** acknowledged.
- More formally: *A liveness property is a property for which any finite path can be extended to a path satisfying the property.*

[Foster et al.: Assertion-Based Design. 2nd Edition, Kluwer, 2010.]

In theory, liveness properties can only be falsified by an infinite simulation run.

- Practically, we often assume that the “graceful end-of-test” represents infinite time.
 - If the good thing did not happen after this period, we assume that it will never happen, and thus the property is falsified.

Introduction to SVA

What is an assertion?

- An assertion is a description of a property of the design
 - If a property that is being checked does not behave the way we expect it to then the assertion fails
 - If a property that is forbidden from happening in a design happens then the assertion fails

```
`ifdef ma
```

```
if (a & b)
```

```
$display ("Error: mutually asserted a and b");
```

```
`endif
```

Types of SystemVerilog Assertions

There are 2 types of Assertion in SystemVerilog

- Immediate Assertions
 - Immediate assertions are procedural statements used mainly in simulation
- Concurrent Assertions
 - Based on clock cycles
 - For example - "A Request should be followed by an Acknowledge occurring no more than two clocks after the Request is asserted."

Concurrent assertions

- Based on clock cycles
- Test expression is evaluated at clock edges based on the sampled values of the variables involved
- Can be placed in a procedural block, a module, an interface or a program definition
- Can be used in both “formal” and “dynamic”

Building Blocks of SVA

1. Create boolean expressions

```
sequence s1;
```

```
@(posedge clk) a ##2 b;
```

2. Create sequence expressions

```
property p1;
```

```
s1;
```

```
endproperty
```

3. Create property

```
a1: assert property(p1)
```

4. Assert property

```
c1: cover property(p1)
```

5. Cover property • • •

**Might be
automatic
in the tool?**

Basic SVA Syntax and Semantics

Clock Definition in SVA

Clock defined in sequence
sequence s1;
 @(posedge clk) a ##2 b;
endsequence;

property p1;
 s1;
endproperty

a1: assert property(p1)

Clock defined in property
sequence s1;
 a ##2 b;
endsequence;

property p1;
 @(posedge clk) s1;
endproperty

a1: assert property(p1)

Best to keep sequences independent of clock

Will increase the sequence re-use

The ## delay operator

- Usage:
 - ## integral_number
 - ## identifier
 - ## (constant_expression)
 - ## [cycle_delay_const_range_expression]
- ## can be used multiple times within the same chain.
 - E.g., a ##1 b ##2 c ##3 d
- Semantics:
 - a ##0 b
 - Sequence overlap: b starts on the same clock when a ends:
 - a ##1 b
 - Sequence concatenation: b starts one clock after a ends
- You can use an integer variable in place of the delay.
 - E.g., a ##delay b

Using a range in the delay operator

- You can specify a range of absolute delays too.
 - E.g., a ##[1:4] b
 - b starts within 1 to 4 cycles of when a ends
- You can also use a range of variable delays.
 - E.g., a ##[delay1:delay2] b

The semantics of “a ##2 b”

- What are the conditions for this to hold?

**Clock defined in property
sequence s1;**

**a ##2 b;
endsequence;**

**property p1;
 @(posedge clk) s1;
endproperty**

a1: assert property(p1)

- **There is a problem with this assertion**
 - It does not say “if a is high then b must be higher 2 cycles later”
 - It says “a is high and b high 2 cycles later” is true on EVERY cycle!
- **How do we assert “b is high 2 cycles after a is high”?**

Implication Operator

- Implication is equivalent to “if-then”
- Left hand side is “antecedent”
- Right hand side is “consequent”
- Antecedent is a gating condition
- If the antecedent does NOT succeed then property succeeds by default: vacuous success
- If antecedent does succeed then consequent is checked

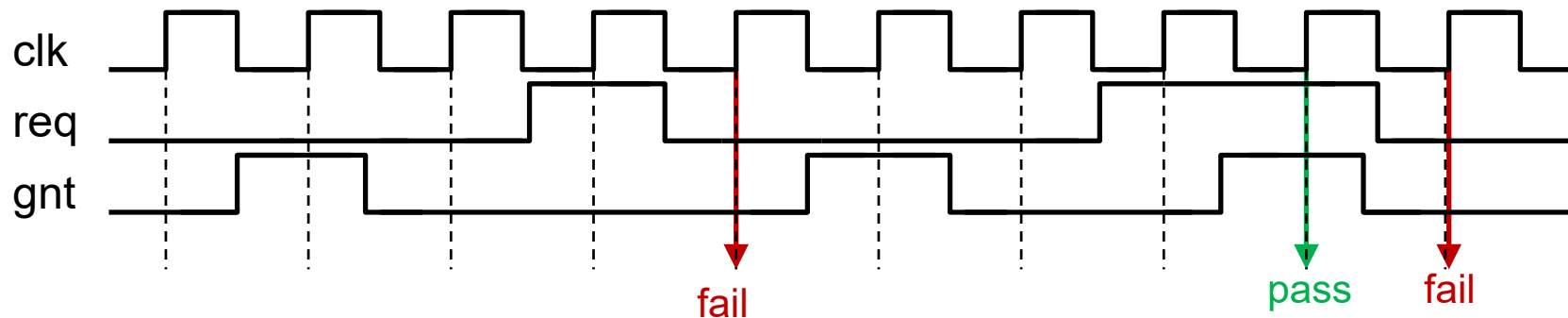
Implications

- Properties typically take the form of an implication.
- SVA has two implication operators:
- \Rightarrow represents logical implication
 - $A \Rightarrow B$ is equivalent to $(\text{not } A) \text{ or } B$,

non-overlapping
implication

where B is sampled one cycle after A.

```
req_gnt: assert property ( req  $\Rightarrow$  gnt );
```



Implications

- SVA has another implication operator:
- $| \rightarrow$ represents logical implication
 - $A | \rightarrow B$ is equivalent to $(\text{not } A) \text{ or } B$,
where B is sampled **in the same cycle as** A.

```
req_gnt_v1: assert property ( req ==> gnt );
```

```
req_gnt_v2: assert property ( req |-> ##1 gnt );
```

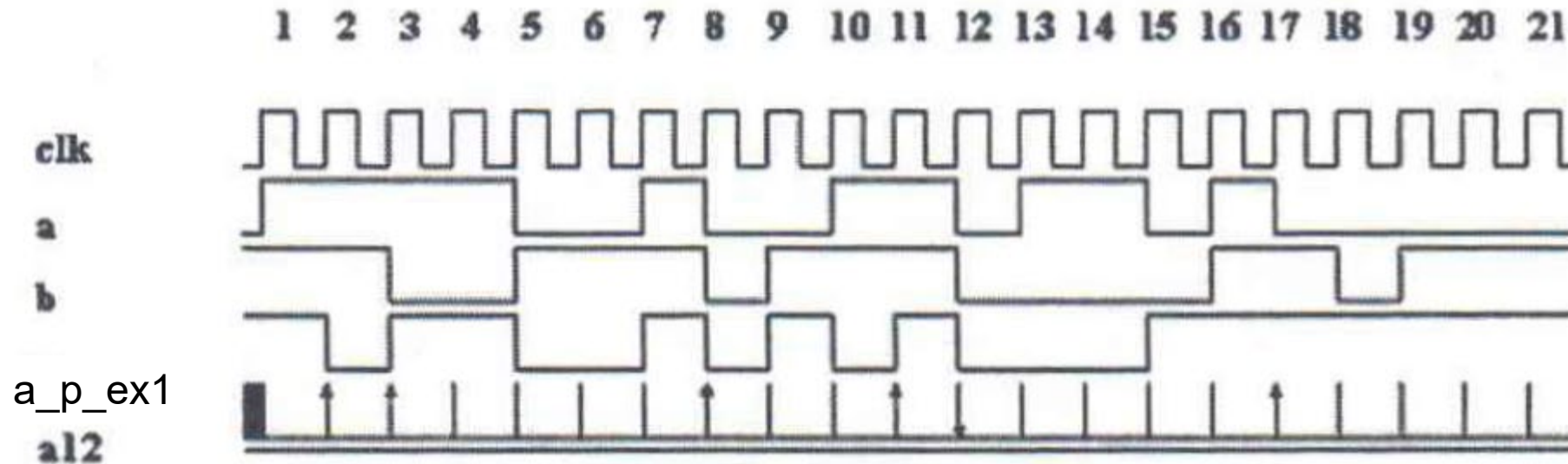
The overlapping implication operator $| \rightarrow$ specifies behaviour in the same clock cycle as the one in which the LHS is evaluated.

Delay operator $##N$ delays by N cycles, where N is a positive integer including 0.

Both properties above are specifying the same functional behaviour.

Timing Windows

- // timing window in SVA
a_p_ex1: assert property(@(posedge clk) (a && b) |-> ##[1:3] c);
- Note:
 - There can ONLY be one valid start on a positive clock edge
 - But there can be MULTIPLE valid endings



Built-in System Functions

- \$onehot(expression) : checks that the expression is one-hot, i.e. one bit of the expression can be high on any given clock edge
- \$onehot0(expression) : checks that the expression is zero one-hot, i.e. one bit of the expression can be high or none of the bits can be high on any given clock edge
- \$isunknown(expression) : checks if any bit of the expression is X or Z

Useful System Verilog Functions for Property Specification

- `$past (expr)`
 - Returns the value of `expr` in the previous cycle.
 - Example:

```
assert property ( gnt |-> $past(req) );
```
- `$past (expr, N)`
 - Returns the value of `expr` `N` cycles ago.
- `$stable (expr)`
 - Returns true when the previous value of `expr` is the same as the current value of `expr`.
 - Represents: `$past (expr) == expr`

SVA with Parameters

```
module generic_chk (input logic a, b, clk);
```

```
    parameter delay = 1;
```

```
    // SVA using parameters
```

```
    property p16;
```

```
        @(posedge clk) a |-> ##delay b;
```

```
    endproperty
```

```
    a16: assert property(p16);
```

```
endmodule
```

```
module simple_seq;
```

```
    logic clk, a, b, c, d, e;
```

```
    .....
```

```
    generic_chk #(.delay(2)) i1 (a, b, clk);
```

```
    generic_chk i2 (c, d, clk);
```

```
    .....
```

```
endmodule;
```

Formal Arguments in a Property

```
property arb (a, b, c, d);  
@(posedge clk) ($fell(a) ##[2:5] $fell(b)) |->  
##1 ($fell(c) && $fell(d)) ##0 (!c&&!d) [*4]  
##1 (c&& d) ##1 b;  
endproperty
```

```
a_arb_1: assert property(arb(a1, b1, c1, d1));  
a_arb_2: assert property(arb(a2, b2, c2, d2));  
a_arb_3: assert property(arb(a3, b3, c3, d3));
```

SVA using local variables

- A variable can be declared locally and
 - Can be assigned to, stored and manipulated

```
property p_local_var;  
int lvar;  
@(posedge clk) ($rose(enable1), lvar = a)  
|-> ##4 (aa == (lvar*lvar*lvar));  
endproperty
```

```
a_local_var: assert property(p_local_var);
```

These are very good for data properties

Formal and Coverage

Coverage in Formal: use of constraints

- First, some background
 - The formal tool will model the design as an FSM
 - The constraints (assumptions) defined will reduce that FSM
 - That is the tool will remove the states that become unreachable under the given constraints
- We need to ensure we do not “over constrain”
 - Otherwise we explore a state space that is too small
 - And we might miss legitimate bugs
- Over constraint in simulation
 - Typically detected by code and functional coverage
- Over constraint in Formal?
 - Covered in the next few slides

Coverage in Formal: implication

- Implication in formal creates a different type of coverage problem
 - Did I hit my antecedent?
- If not
 - Then we have a vacuous proof of the implication
- We need to consider this differently to over constraint!
- The following slides discuss
 - Over constraint
 - Vacuous implication proofs

Coverage in Formal

- Cover Properties
 - Used to avoid vacuous proofs in implications
 - Do we actually see a completing sequence for the antecedent so we get into the Enabled state
- Design coverage
 - Looks at how much of the FSM is explored,
 - and thus how much of the RTL code was explored
 - this uses the coverage app

Coverage in Formal: Design Coverage

- Looks at how much of the FSM is explored,
 - and thus how much of the RTL code was explored
- Coverage metrics used
 - Code
 - Line, branch, expression, toggle
 - Functional
 - Using the SV “cover” directive

Connecting SVA to the design

Two methods for connecting checkers to the design:

1. Embed on inline the checkers in the module definition
2. Bind the checkers to a module, an instance of a module or multiple instances of a module

```
bind <module_name or instance_name>  
<checker name> <checker instance name>  
(design signals)
```

Lab Time

Dealing with Complexity

Property Checking – Outputs from the tool

- Proved ✓
 - Increase in confidence
- Failed(n) ✓
 - We found a bug
 - Or an under constraint!
 - Or a badly written property!
- Explored(n) ?
 - What do we do now?

Overcoming Complexity Issues - Abstraction

- Some constructs are complex for formal tools
- Instead, we can use abstraction
 - create a model which resembles reality
 - but with much less detail.
- Successful formal verification of large designs may require that parts of the design are abstracted.
 - Learning how and where to apply abstractions will result in more proven properties and more bugs found.
- This is a big topic that is only partially covered here

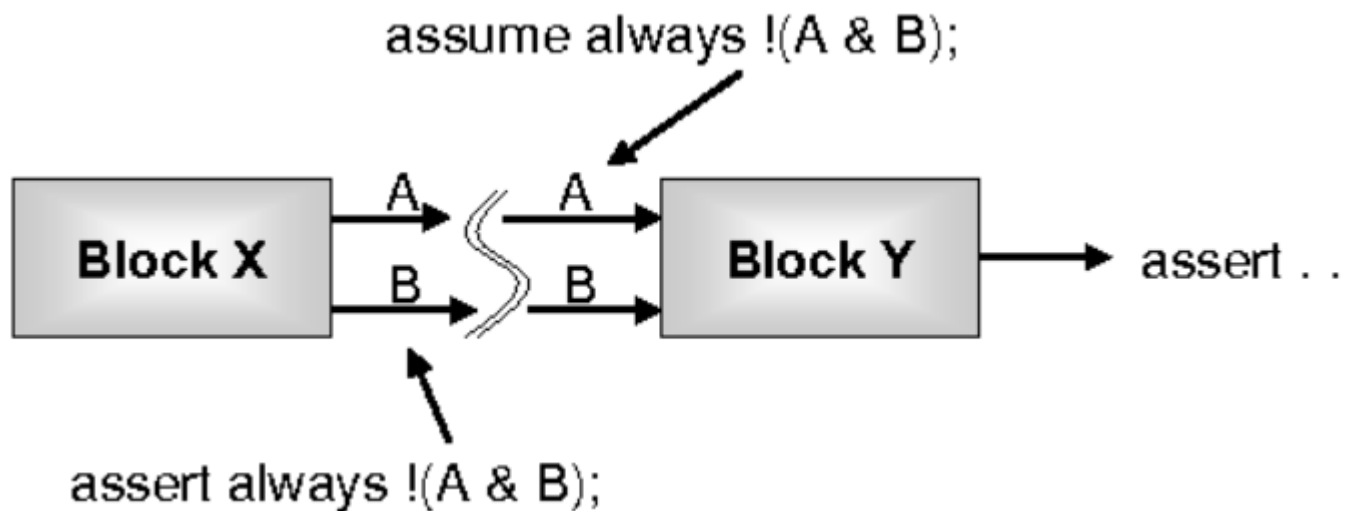
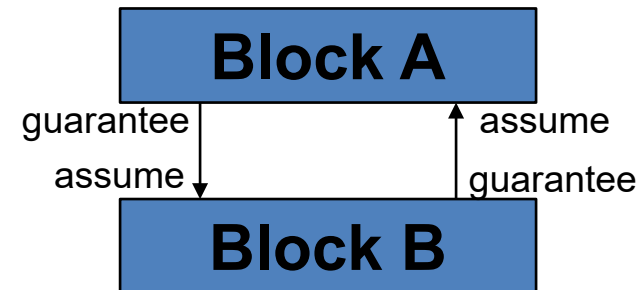
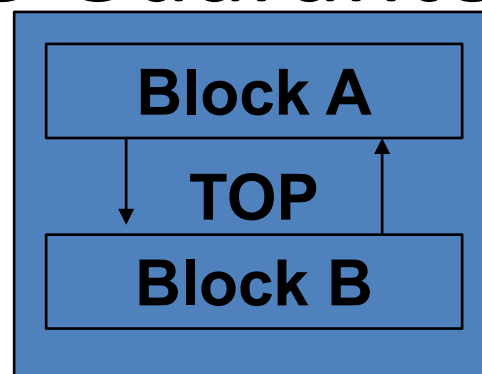
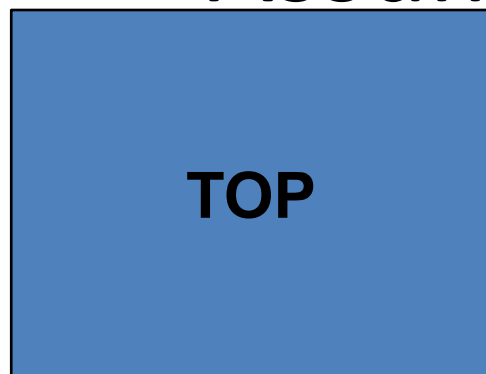
Counters

- Counters are often used to trigger events
 - E.g. a timeout
- But counters add complexity for formal
 - They add sequential depth
 - N-bit wide add $2^{**}N$ cycles to timeout
- But we only 3 interesting states
 - Initial state, 0
 - Intermediate values between 1, .., $2^{**}N - 1$
 - Max value $2^{**}N$
- We can model this as a very simple FSM
- Some tools might do automatically

Formal helpers

- Mutations
 - RTL changes to reach corner-cases in fewer cycles (e.g. FIFO reduction). Used in simulation too. Non-deterministically enabled in formal
- Initial value and other abstractions
 - Skip “configure and populate” cycles to reach interesting cases faster
 - Skip irrelevant logic

Assume Guarantee Paradigm



Formal in the Design Flow

The Strengths of Property Checking

- Ease of set-up
 - No test bench required, add constraints as you go, VIP?
- Flexibility of verification environment
 - Constraints can be easily added or removed
- Full proof
 - Of the properties under the given constraints
 - (Can also prove “completeness” of the properties)
- Intensive stressing of design
 - Explored(n) constitutes a large amount of exploration of the design
 - Judgement when the number of cycles explored in a run is sufficient
 - Significant bugs already found within a this number of cycles
- Corner cases
 - Find any way in which a property can fail (under the constraints)

Potential issues with formal verification

- False failures
 - Need constraints to avoid invalid behaviour of inputs
- False proofs
 - Bugs may be missed in an over-constrained environment.
- Limits on size of the model that can be analysed
- Non-exhaustive checks: *Explored(n)*
 - Interpret the results
 - Can require significant knowledge and skill
- Non-uniform run times
 - Often it cannot be predicted how long it will take for a check either to terminate or to reach a useful stage

This can make formal unpredictable!

A Taxonomy of Methodologies

- Bug avoidance
 - Improve quality before any property checks are run
 - Visualization
 - Clarification of spec
- Bug hunting
 - Use model checking to look for bugs
 - Do not worry if proofs do not complete
- Bug absence
 - Aim to ensure that properties are fully proven
 - Aim to get a “complete” set of properties
- Bug analysis
 - For bugs in FPGA prototypes or in Silicon
 - It may be hard to recreate the conditions that causes a bug
 - By writing the symptom of the bug as a property, one can generate a waveform that can be analysed

Design bring-up

- **Aid for design during RTL development**
 - Verification test benches may not be ready
 - Designers write “throw-away” test benches
- **Formal for designers**
 - Getting a simple working formal setup is relatively fast
 - Write the constraints
 - Write basic properties
 - Check the RTL is not completely broken
 - Check assumptions on signal properties and equivalence
 - Investigate or visualise sequences/scenarios
 - Cover “set error bit” “generate interrupt signal”

```
cover property (@posedge clk (empty |-> ##[0:$] full));
```

```
cover property (@posedge clk (full |-> ##[0:$] empty));
```

- **Catch bugs early**
 - Formal counter-examples shorter to debug than simulation failures

Bug analysis using Formal

- For example
 - A bug found late in the design process
 - Difficult to hit in simulation
 - Found by human review
 - Observed in the field
- Investigate around a specific bug
 - Reproduce bug in formal
 - Write a suitable formal environment and property
 - Find similar bugs
- Check bug fixes

Formal “apps”

- Superlint (Autochecks)
- X-propagation
- Clock domain crossing
- Clock-gating
- Protocols
- Embedded assertions
- FSM
- SEC
- System registers
- Coverage Closure

Superlint (Autochecks)

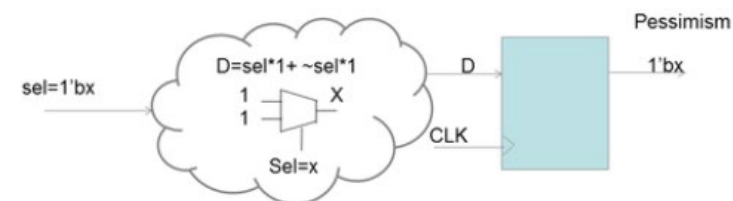
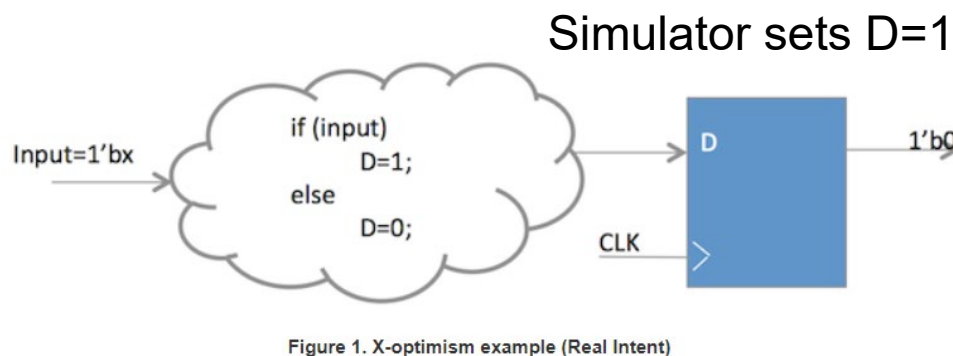
- Check assertions for:
 - Overflows
 - Out-of-bound indexing
- Automatically generated
- Waiver mechanism is mandatory
- Meticulous lint tool

Protocols

- Certify compliance with standard protocols
 - AXI, ACE, AHB, ATB, APB
- Protocol checkers integrated into EDA solutions
 - Can be used as master or slave
 - Highly configurable
 - The properties are optimized for formal rather than simulation

X-propagation

- Detect and debug X-propagation issues on RTL
- Simulators do not deal correctly with X's
- This has become a bigger issue in recent years because of the use of power-gating architectures



'if-then-else' or 'case' statements

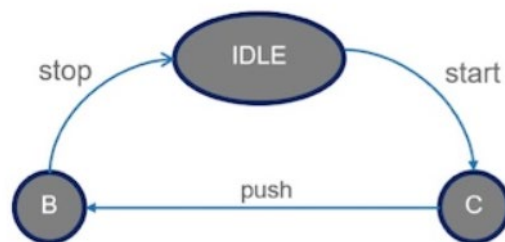
The X state will not satisfy the logic test, the block will be assigned the default case. This may convert the X to a 'known' value or propagate it further into the simulation, masking a bug

Data_2	Data_1	Sel	Simulator	Hardware
0	0	X	0	0
0	1	X	X	X
1	0	X	X	X
1	1	X	X	1

Simulation is inaccurate in the presence of X's

Finite State Machines

- What can go wrong with finite state machines?
 - Deadlock: once the FSM has entered a particular state, there is no valid input that will trigger its exit from that state.
 - Unreachable states are created when there is no combination of inputs that will lead to that state.
- Automatic generation of properties
 - State reachability
 - Transition conformity
- Simple textual FSM specification
 - States
 - Transitions
 - Automatically translate into properties for proof of implementation



```
assert property (@(posedge clk) (state == IDLE) && start ==> (state == C));  
assert property (@(posedge clk) (state == C) && push ==> (state == B));  
assert property (@(posedge clk) (state == B) && stop ==> (state == IDLE));
```

Figure 2. A state machine and assertions in SVA (OneSpin Solutions)

Formal in the organisation

Strategic Issues with Formal

- What simulation do I replace?
 - Short answer is none unless block is done completely formally
 - The metrics are too different
- We don't know if or when it will complete
 - Formal can take a long time to give very poor results
- A high level of skill might be required
 - To write the correct properties and constraints
 - To drive the tools
 - And to drive into bug avoidance in the future
- So why bother?
 - You can “get it for free” on the back of assertion-based verification
 - There are requirements that cannot be verified through simulation
 - Cache coherency, liveness, deadlock,...
 - We need it to cope with the increasing complexity of verification

So how do I get started with Formal Verification

- Targeted applications
 - Coverage closure, X-propagation, etc
 - Easy to apply but not of significant value
- Get designers to use it
 - Write assumes, coverage and properties that can be re-used
- Real exploitation requires strategic investment
 - Training for writing “bug hunting” properties
 - Standardise on when, where and how to write
 - Automation of the flows
- Create bug absence experts
 - Requires careful selection and training
 - Centralise the skills?
 - These people will also be good at bug analysis
- Bug avoidance is a longer term goal

The main EDA Tools

Cadence Jasper: Best-in-class Formal by far

- Formal is a mainstream verification technology
- Formal is growing rapidly in the verification mix: complementary to simulation
- Industry's leading formal technology is JasperGold from Cadence













Formal Scalability Leadership =

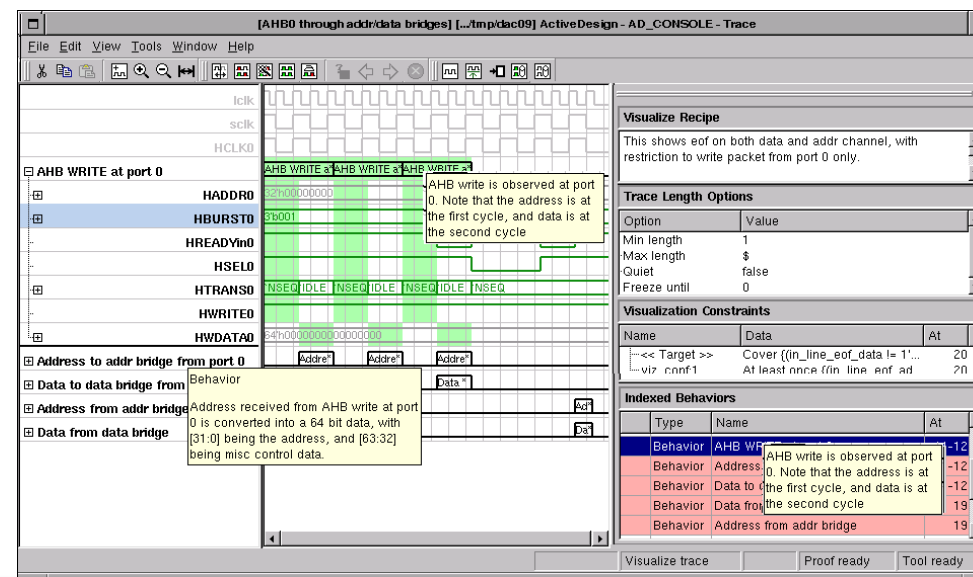
- more verification
- in less time
- on bigger designs

JasperGold verification platform

Solve specific verification problems
with targeted JasperGold® Apps

Highly interactive formal debug
transforms to fit the App

 Formal Property Verification App	 SuperLint (AFL) App	 Design Coverage Verification App	 Sequential Equivalence Checking App
 X-Propagation Verification App	 CSR Control/Status Register Verif. App	 Connectivity Verification App	 UNR Coverage Unreachability App
 Clock Domain Crossing App	 Functional Safety Verification App	 Low Power Verification App	 Security Path Verification App



Broad formal engine and infrastructure

Assertion Based Verification IPs for AMBA and other common protocols

Programmable Interface via TCL

ProofGrid™ Manager assigns best engine for task

JasperGold 2018.09 / 2018.12 milestone releases

Comprehensive Formal Signoff

Engine-independent coverage measurement

All-new intuitive formal coverage analysis

Advanced Design Scalability

Compiles bigger designs faster

Up to 70% memory reduction: uses smaller servers

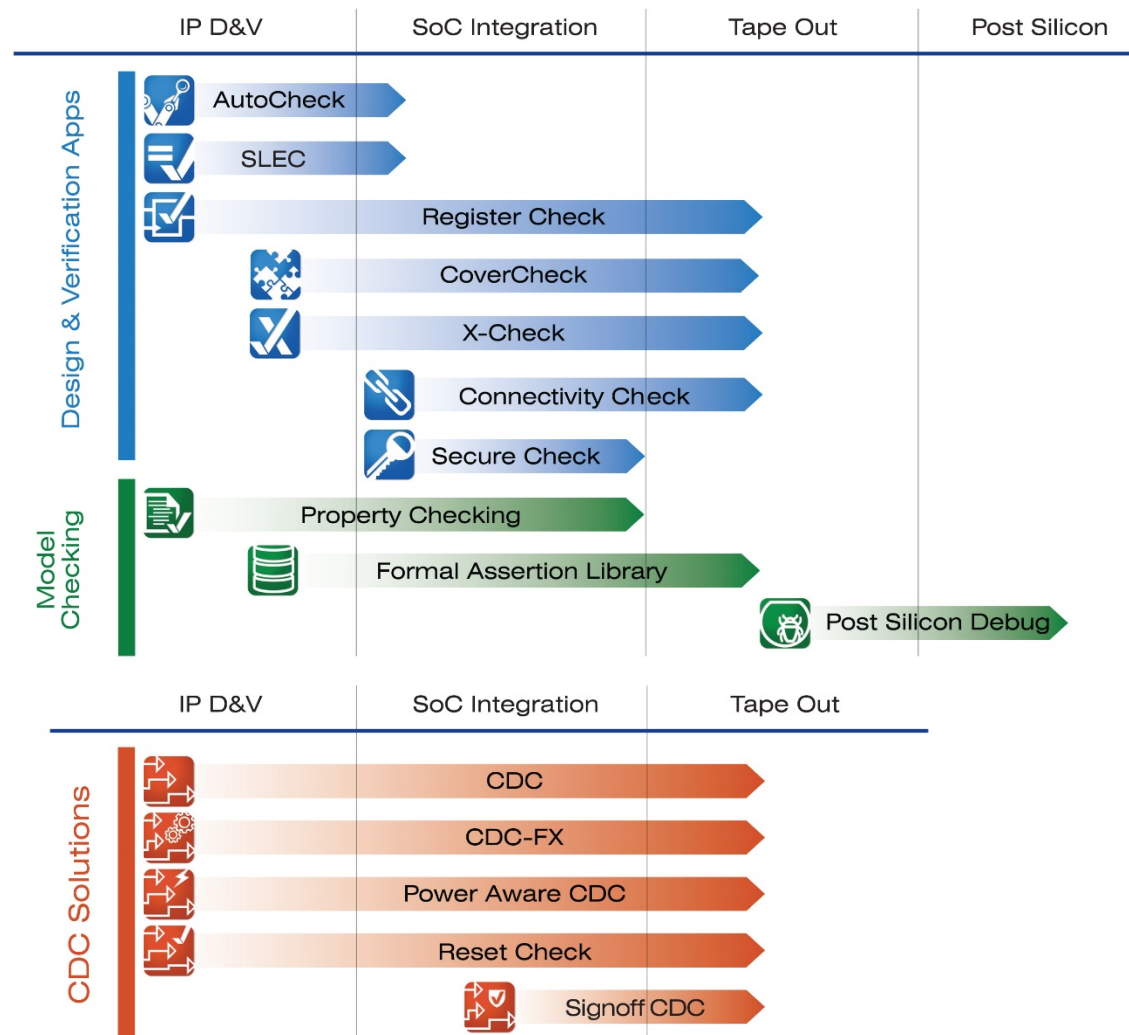
Smart Proof Technologies

Big increases in performance & convergence

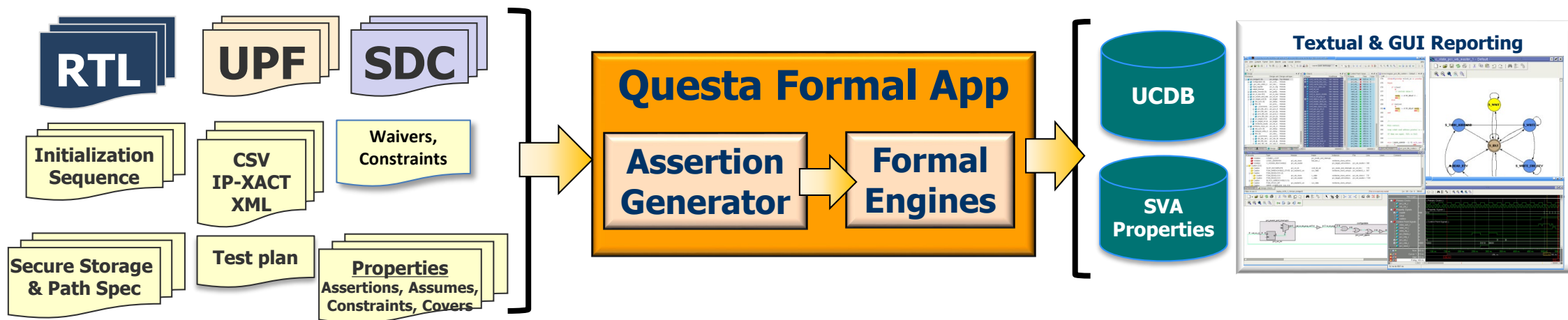
Uses Machine Learning for out-of-the-box proofs & regression

Mentor's Formal Apps Deliver Automated, Exhaustive Verification For Every Project Phase

- Formal-based apps focus on specific, high-value verification challenges; from IP to SoC levels
- Apps auto-generate assertions, saving countless hours of work
- Because formal is exhaustive, a formal app is THE best tool for the corresponding task
- Results can be integrated with simulation and verification planning and management



Mentor: How Do Formal Apps Work?



Inputs

RTL + Task-Related files



















Processing

Assertion generator
+ formal engines

Outputs

Waveforms,
Text&GUI Report(s),
Properties, UCDB

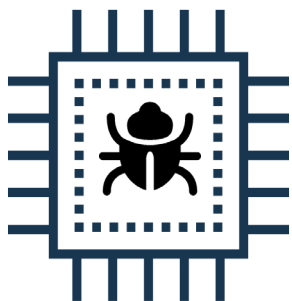
Mentor: Automated Formal Apps Fix Expensive, Painful Problems

Pain	FPGA user value	Mentor's Solutions
Needing to wait for the UVM TB before "serious" verification can begin		 AutoCheck  PropCheck
Finding corner-case bugs very late, when they are harder to fix		 PropCheck  Register Check
Confirming customizations to AMBA bus protocol didn't go too far		 AMBA Formal Assertion Library
Code Coverage closure & dead code analysis		 CoverCheck
Register policy corner cases hard to find with simulation		 Register Check
SoC and pad ring static & dynamic internal connectivity; No connectivity spec for legacy IPs		 Connectivity Check
Is there an unintended HW backdoor to secure/safety critical paths & storage?		 Secure Check
Erratic HW failure from 'X' – low power or post-reset		 X Check
Verify absolute sequential equivalency between RTL IPs (ECO, Low Pwr, Fault)		 SLEC
Burning opportunity cost trying to root-cause a post-silicon bug		 Post Silicon Debug
Multiple clock and reset domains cause metastability that hang the chip		 CD  PA  CD  X, F  Sign  CDC

OneSpin Solutions

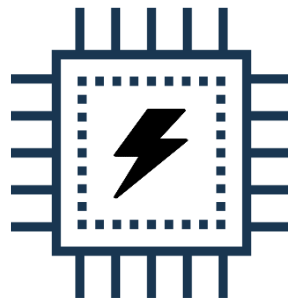


Functional Reliability



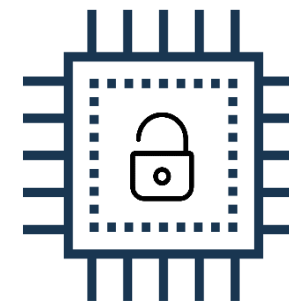
+

Functional Safety



+

Trust & Security



Addressing IC Integrity Challenges

Design Exploration
Protocol Violations
Integrate Formal/Sim Coverage
End-to-End User Assertions
HLS/SystemC Verification
Synthesis/P&R Errors

FMEDA Support
Excessive Fault Simulation
Insufficient Diagnostic Coverage
Incorrect Safety Mechanisms
ISO 26262 Compliance
DO-254 Compliance

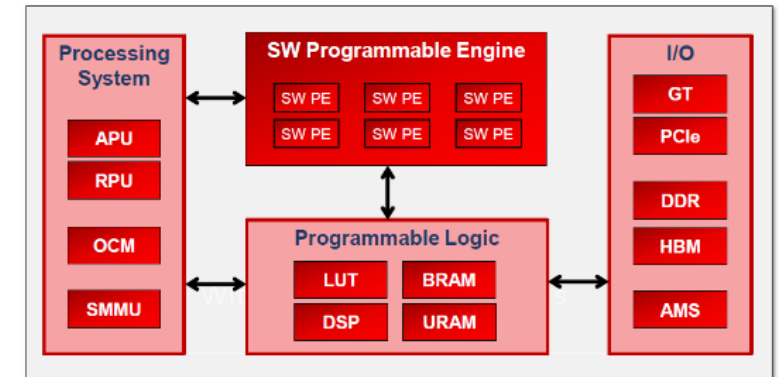
Denial of Service
Data Leakage
Privileges Escalation
Data Integrity/Confidentiality
Hardware Backdoors
Hardware Trojans

OneSpin – AI, ML, 5G, RISC-V



Heterogeneous computing hardware platforms

- Top-level connectivity verification supporting XL chips
 - 1M+ connections, 60M+ module instances, 30K+ modules
 - Abstract connectivity specification expanded by tool
- Floating-point unit (FPU) automated verification
- Coherent accelerators protocol compliance
- HLS flow support (SystemC/C++)
- Reliable synthesis and P&R implementation flows
 - Support for Intel-Altera, Xilinx, and Microsemi devices



RISC-V

- ISA and privileged ISA formalization using SystemVerilog Assertions
- Unbounded proofs, 100% proven functional coverage



OneSpin – Functional Safety



Automotive, ISO 26262 compliance

- Computation of safety metrics: SPFM, LFM, PMHF
- Minimize or replace fault simulation
- Verification of safety mechanisms
- Tool qualification kit certified by TÜV SÜD



Avionics, DO-254 compliance

- Minimize or replace gate-level simulation
- Equivalence checking to verify advanced FPGA optimizations
- Speed-up elemental analysis
- Tool qualification kit



Nuclear, railway, medical, industrial



Further Reading

- SVA
 - [http://s1.nonlinear.ir/epublish/book/SVA The Power of Assertions in SystemVerilog 9783319071381.pdf](http://s1.nonlinear.ir/epublish/book/SVA%20The%20Power%20of%20Assertions%20in%20SystemVerilog%209783319071381.pdf)
- Abstraction
 - <http://www.techdesignforums.com/practice/technique/the-art-of-abstraction/>
- Writing formal VIP
 - <https://www.design-reuse.com/articles/20327/assertion-ip-formal-verification.html>
- Writing a formal verification test plan
 - [https://www.researchgate.net/publication/228360702 Guidelines for creating a formal verification testplan](https://www.researchgate.net/publication/228360702_Guidelines_for_creating_a_formal_verification_testplan)
- Under the hood (???)
 - [https://en.wikipedia.org/wiki/Boolean satisfiability problem](https://en.wikipedia.org/wiki/Boolean_satisfiability_problem)

Further reading

- Good T&VS conference papers
 - Alex Orr, Princip, Broadcom al Engineer – IC Design
 - “My first 100 days in formal-land”
 - <https://www.testandverification.com/conferences/formal-verification-conference/formal-verification-conference-2015/>
 - Better Living Through Formal
 - <https://www.testandverification.com/conferences/formal-verification-conference/fv2016/better-living-through-formal/>
 - Prof. Ashish Darbari, Leader of Advanced Verification Methodology Group, Imagination Technologies Limited **“The Ten Myths About Formal”**
 - <https://www.testandverification.com/conferences/formal-verification-conference/formal-verification-conference-2015/speaker-professor-ashish-darbari-imagination-technologies/>

Further reading

- SNUG Austin 2018
 - Formal Property Checking Applied to Low-Power Microcontroller Designs
 - Alan Carlin, Nemo Zhong, NXP Semiconductors Austin, TX USA
 - Tareq Altakrouri, Synopsys Plano, TX USA

Further Work

- Get the labs
 - Email it@testandverification.com
- Any questions
 - Email mike@testandverification.com