Functional Formal Verification

From Finding Bugs to Proving Correctness

Credits:
Parts of these lecture notes are based on a set of slides kindly donated by Dr Yaron Wolfsthal.
[Manager, Formal Methods, IBM Haifa Research Laboratories]

Overview

Functional Formal Verification: Motivation & Taxonomy

Property Verification

- Model checking
- Formal language: PSL/Sugar
- The pragmatic view
- Commercial Tools
  - Experiences using IBM RuleBase

Example: Another Buffer ;)

Future Developments

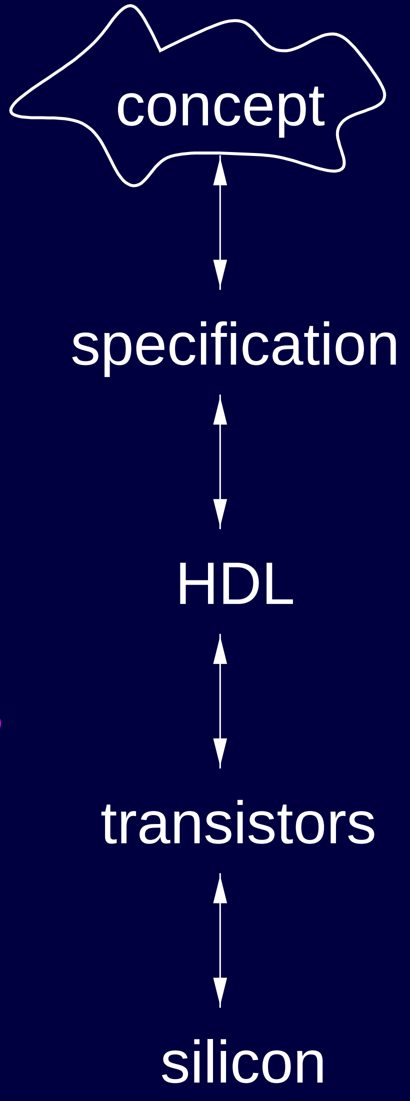
Taxonomy of Verification

What you specified is what you intended?
functional verification - spec level

What you designed (RTL) is what you specified?
functional verification - impl level

What you taped-out is what you designed in RTL?
equivalence checking

What was manufactured is what you taped-out?
(post) production testing



Goal: Eliminate HW design defects before a product is shipped.

Motivation for Functional Formal Verification

Engineers build mathematical models of systems so they can predict their properties a priori through the power of calculation.

It’s the power of mechanized calculations (e.g. fluid dynamics, finite element analysis) that makes the construction of complex physical systems possible.

Formal Methods apply these ideas to the complex logical design of computer systems.

Methodology:

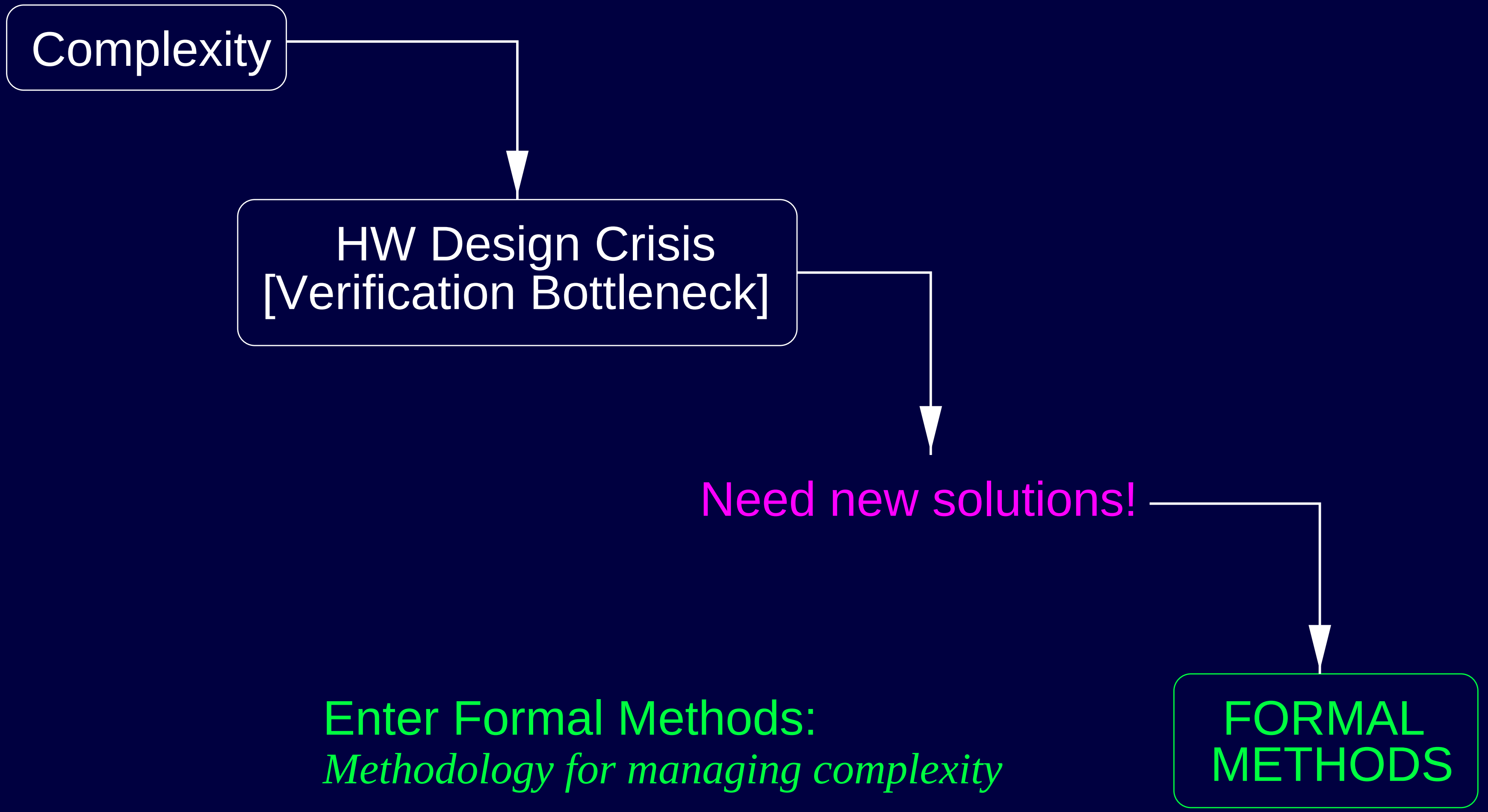
- Build a formal mathematical model of (some aspect of) the system
- Calculate whether or not it has some desired property.

Historical Perspective of Verification Approaches

- In the beginning there was **Simulation** (event- or cycle-based, even with *clever* stimulus generation).

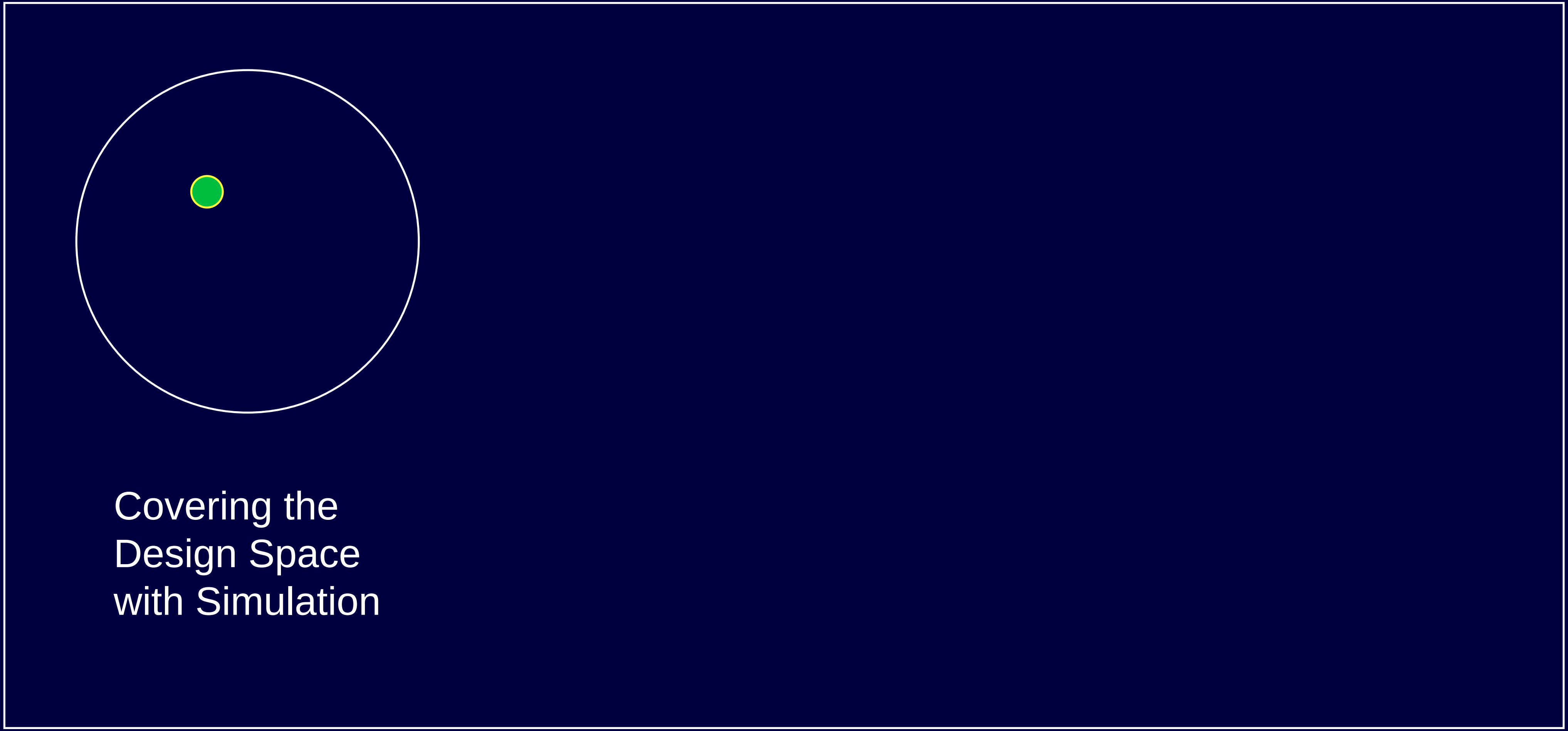
- **Simulation alone has inherent limitations!**

Need for Methodology Shift observed in Industry

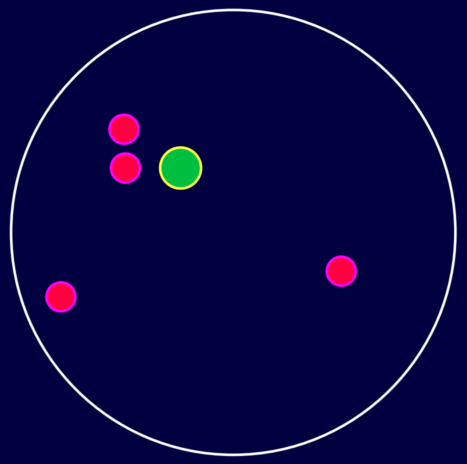


Enter Formal Methods:
Methodology for managing complexity

Formal Verification vs Simulation

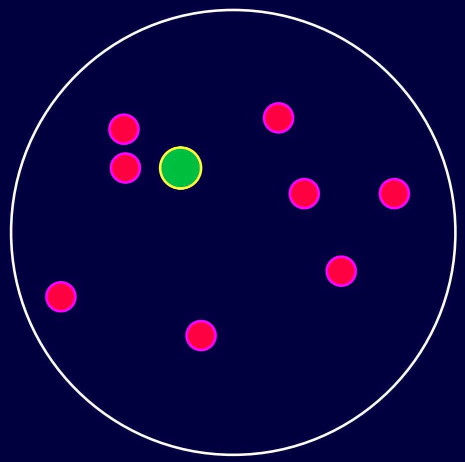


Formal Verification vs Simulation



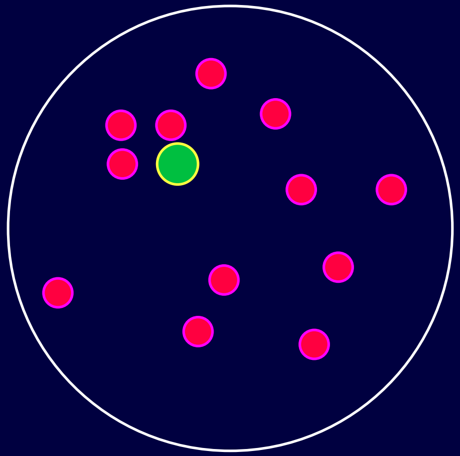
Covering the
Design Space
with Simulation

Formal Verification vs Simulation



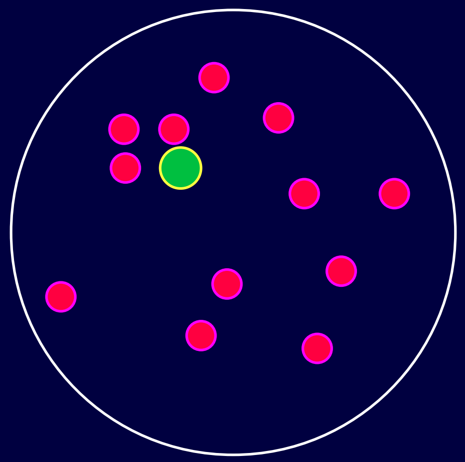
Covering the
Design Space
with Simulation

Formal Verification vs Simulation

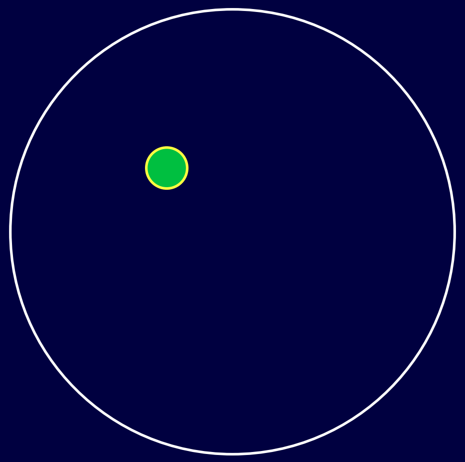


Covering the
Design Space
with Simulation

Formal Verification vs Simulation

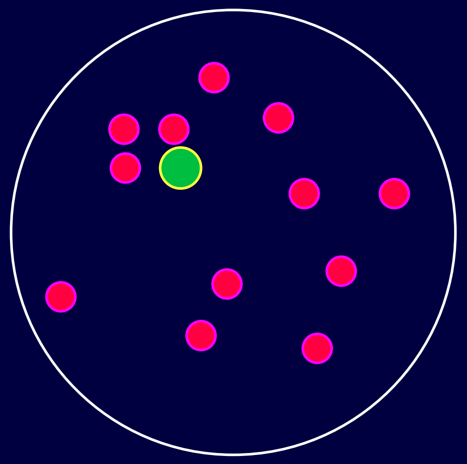


Covering the
Design Space
with Simulation

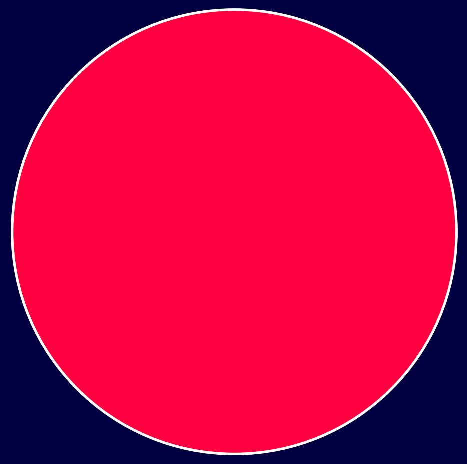


Covering the
Design Space
with Formal
Verification (I)

Formal Verification vs Simulation

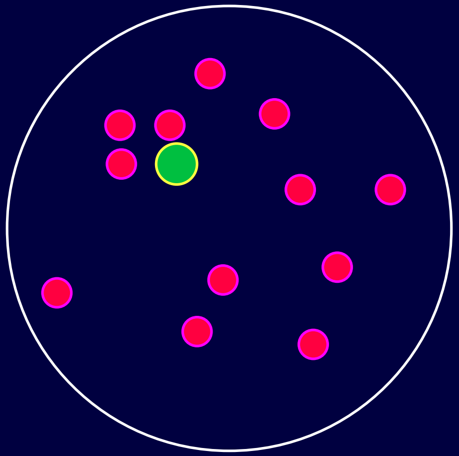


Covering the
Design Space
with Simulation

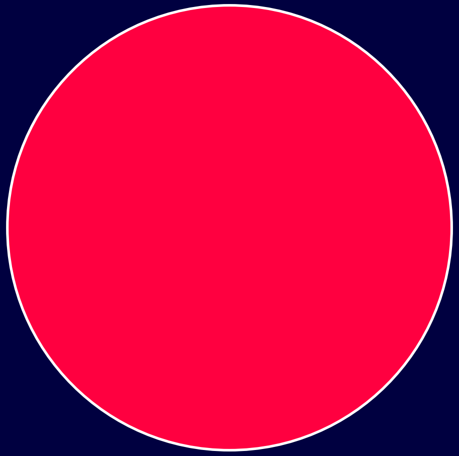


Covering the
Design Space
with Formal
Verification (I)

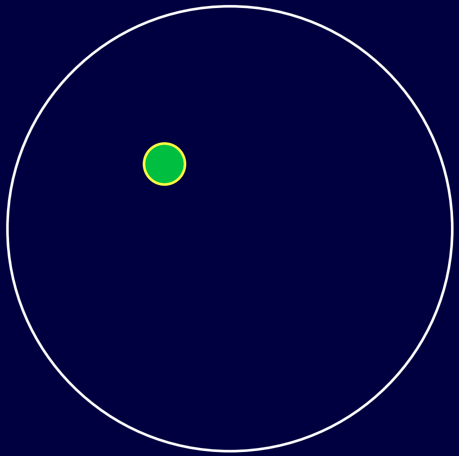
Formal Verification vs Simulation



Covering the Design Space with Simulation

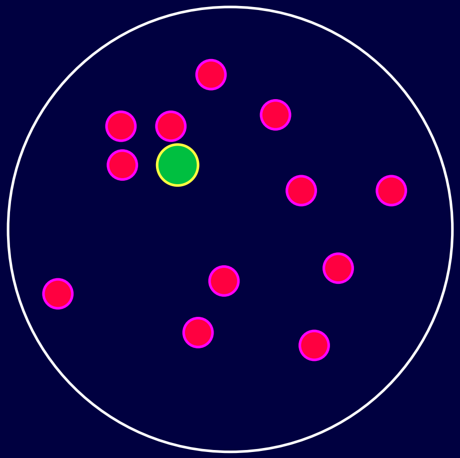


Covering the Design Space with Formal Verification (I)

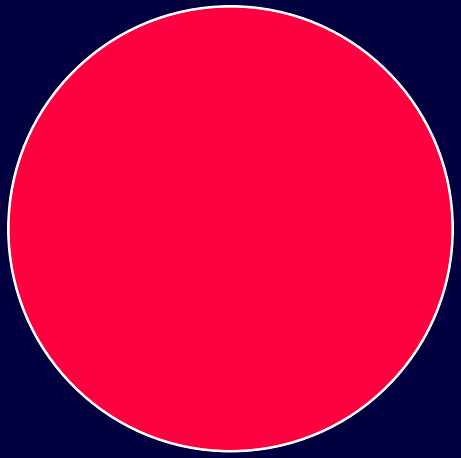


Covering the Design Space with Formal Verification (II)

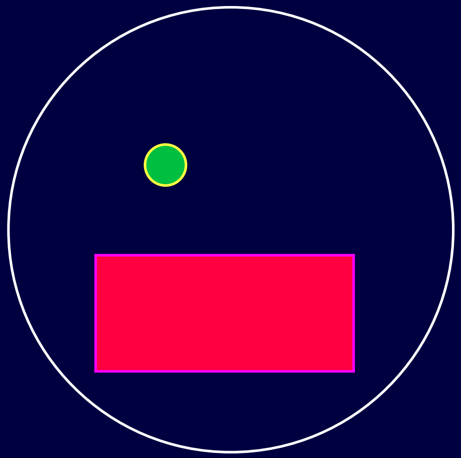
Formal Verification vs Simulation



Covering the
Design Space
with Simulation

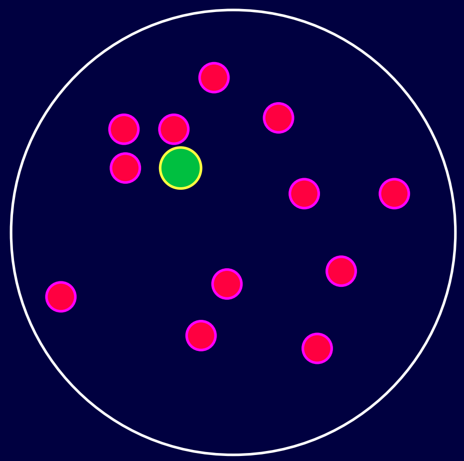


Covering the
Design Space
with Formal
Verification (I)

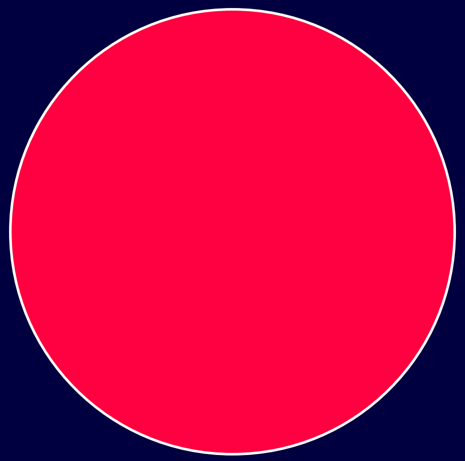


Covering the
Design Space
with Formal
Verification (II)

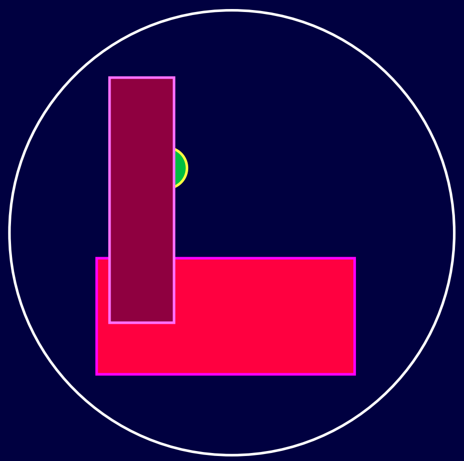
Formal Verification vs Simulation



Covering the Design Space with Simulation

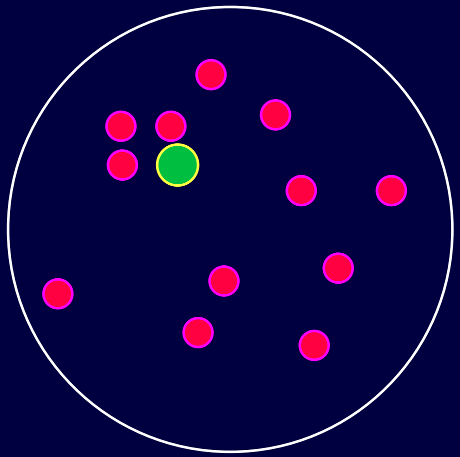


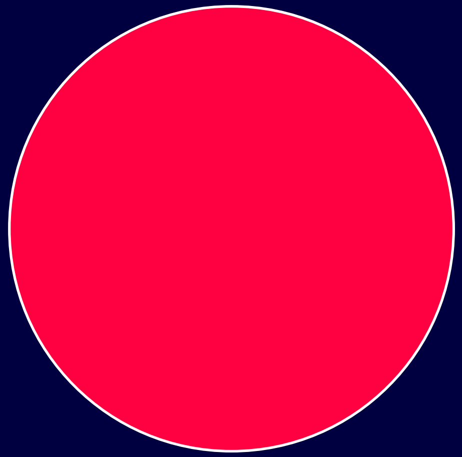
Covering the Design Space with Formal Verification (I)

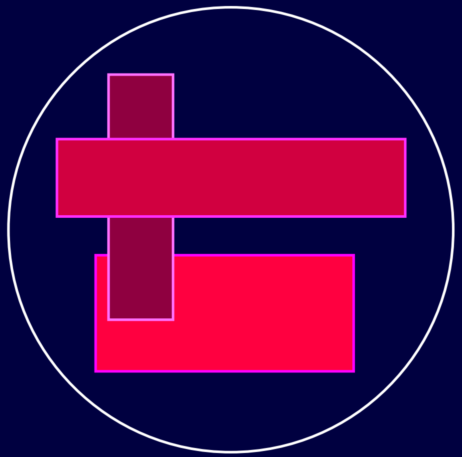


Covering the Design Space with Formal Verification (II)

Formal Verification vs Simulation


Covering the Design Space with Simulation


Covering the Design Space with Formal Verification (I)


Covering the Design Space with Formal Verification (II)

Formal Methods are static: No set-up of test cases!

They establish correctness in a comprehensive way.

The Property Verification Paradigm

Assertion Based Verification (ABV)

Conceptually consists of two major elements

- **Property Specification:**
  - Using a *language* for formally specifying functional requirements and behaviours of a DUV.
    - interface protocols
    - performance constraints
- **Analysis:**
  - Using a *procedure* for establishing that the requirements (properties) hold.

NOTE: Property verification (in general) is not limited to Formal Verification! (See previous lecture on “Assertion Based Verification”!

Sample Properties for Design Verification

- Temporal relationships between signal values
- External and internal protocols
- Cache coherency/consistency protocols

- Mutual exclusion
- Absence of contention/deadlock (similar for livelock)

- Correctness of arithmetic computations

Model Checking - The Background Theory

Model checking is an **automatic model-based verification** approach.

It is based on *temporal logic.*

- **Basic idea:**
  - A formula is not statically true or false in a model (as in propositional or predicate logic).
  - The model contains several states.

  💡 The formula can be true in some states and false in others.

In model checking, the *models* M are *transition systems* and the *properties* ϕ are *formulas in a temporal logic.*

Model Checking Approach

To verify that a DUV satisfies a required property, we must:

1. Construct a model M of the DUV
   - finite state machine (FSM) of spec/DUV

   (– in language of a model checker)
2. Code the property ϕ using the property specification language of the model checker.
3. Run the model checker with inputs M and ϕ.

The model checker answers *yes* if $M \models \phi$ and *no* otherwise.

- If *no* is the answer, most model checkers will produce a *trace of DUV behaviour which causes the failure*, i.e. a **counter example**.

Temporal Logic

Classified according to view of time.

- **Linear-time:** Time is a chain of time instances.
- **Branching-time:** Several alternative future worlds at any given point in time.

Another criteria is whether we think of time as **continuous** or **discrete.**

Examples:

- *Computation Tree Logic (CTL)* [Clarke and Emerson]: Time is branching and discrete.
- *Linear Time Logic (LTL)*: Time is linear and discrete.

Models for Model Checking

Model checking is the process of computing an answer to the question if $M, s \models \phi$ holds, where ϕ is a formula of some logic, M is a model, s a state of that model and $\models$ the underlying satisfaction relation.

Models are often abstractions that omit lots of real features.

Fundamental constituents of models are *states*.

- They could be current values of variables or actual states of DUV.

Second constituent are the *state transitions*.

- Transitions form a binary relation $\rightarrow$ over states S: $\rightarrow \subseteq S \times S$

■ Assignment $x := x + 1$ represents state transition from state s to state s', where the latter state stores the incremented value of x.

Models for Model Checking

Model checking is the process of computing an answer to the question if $M, s \models \phi$ holds, where ϕ is a formula of some logic, M is a model, s a state of that model and $\models$ the underlying satisfaction relation.

- **Models are often abstractions that omit lots of real features.**

Can make it hard to interpret a counter example!

Fundamental constituents of models are *states*.

- They could be current values of variables or actual states of DUV.

Second constituent are the *state transitions*.

- Transitions form a binary relation $\rightarrow$ over states S: $\rightarrow \subseteq S \times S$

■ Assignment $x := x + 1$ represents state transition from state s to state s', where the latter state stores the incremented value of x.

Temporal Operators: X (next time), F (eventually or “in the future”), G (always or “globally”)

$M, s \models EG\phi$

- Holds if there *Exists* a path beginning at s such that ϕ holds *Globally* along the path.

$M, s \models EF\phi$

- Holds if there *Exists* a path beginning at s such that ϕ holds in some *Future* state along the path.

$M, s \models AG\phi$

- Holds if for *All* computation paths beginning at s, ϕ holds *Globally*.

Microwave Oven Specification

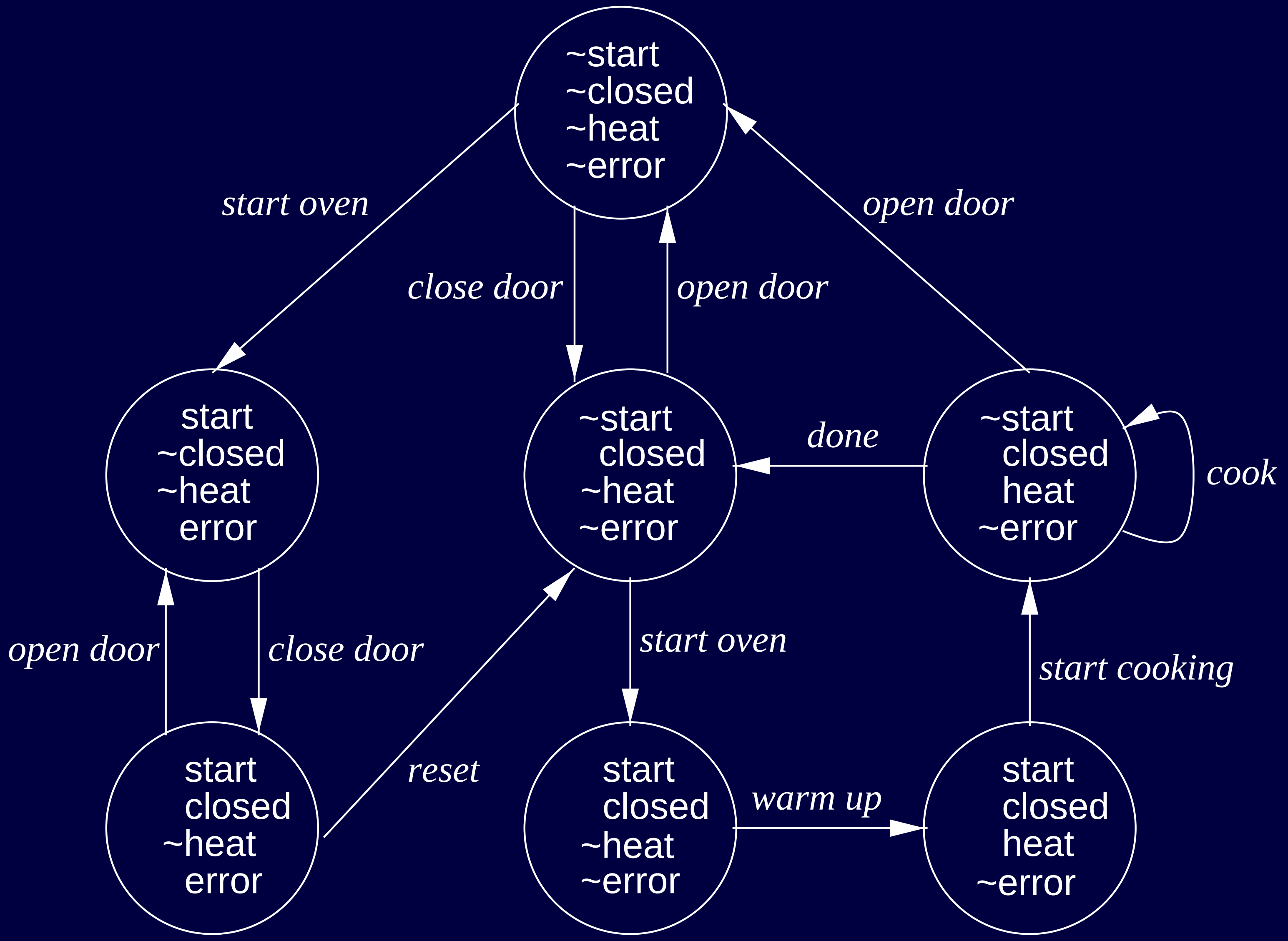
Informal specification of operation instructions for microwave oven:

To cook food in the oven, open the door, put the food inside, and close the door. Do not put metal containers in the oven. Press the start button. The oven will warmup for 30 seconds, and then it will start cooking. When the cooking is done, the oven will stop.

The oven will stop also whenever the door is opened during cooking.

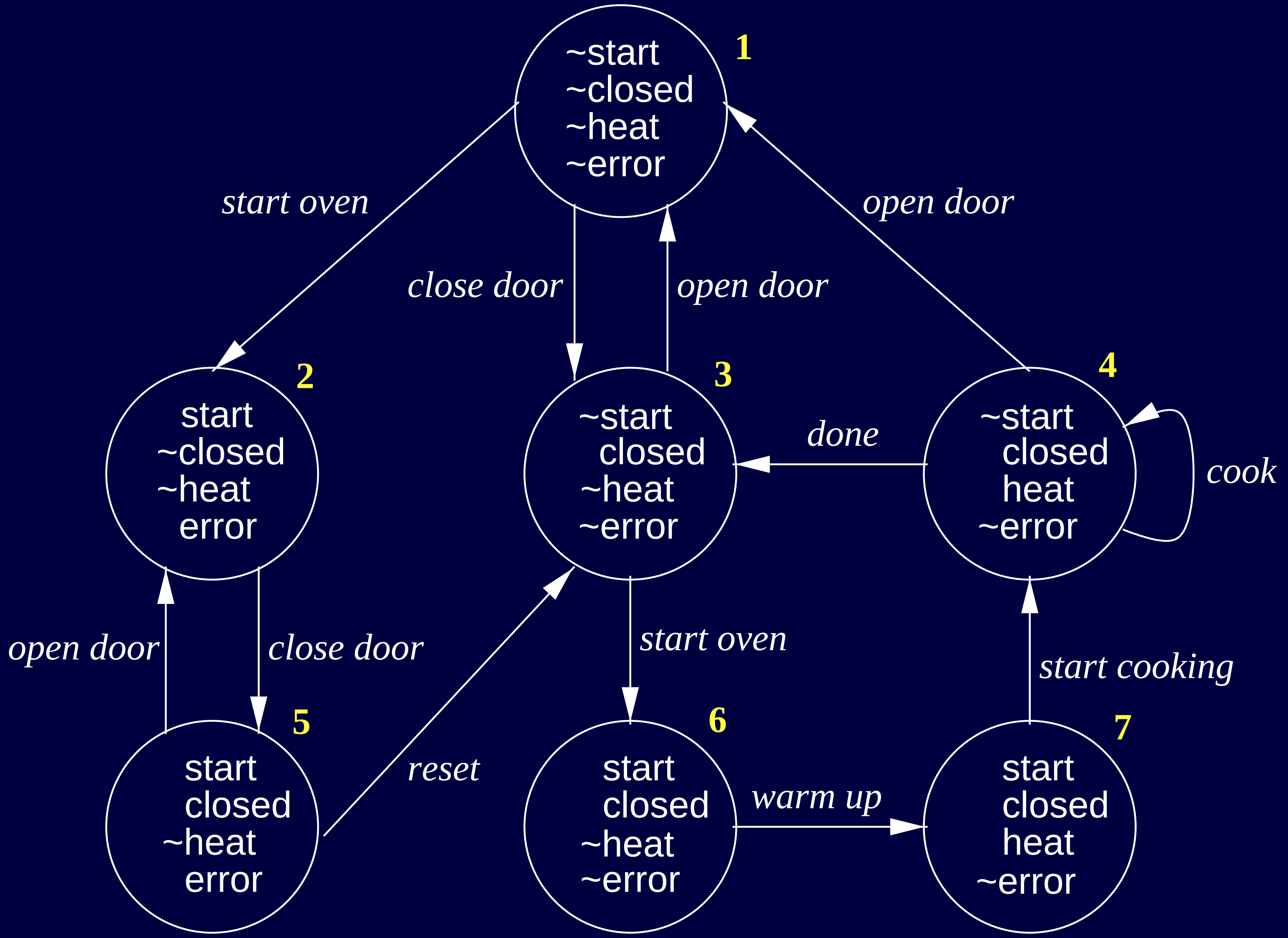
If the oven is started while the door is open, an error will occur, and the oven will not heat. In such a case, the reset button may be used.

Example Model: Microwave Oven



[Source: Edmund M. Clarke, Jr., Orna Grumberg, Doron A. Peled. "Model checking"]

Example Model: Microwave Oven



[Source: Edmund M. Clarke, Jr., Orna Grumberg, Doron A. Peled. “Model checking”]

Check whether *“Whenever the oven is started, then eventually the oven will heat.”* is satisfied.

$$S(start) = \{2, 5, 6, 7\}$$
$$S(\sim heat) = \{1, 2, 3, 5, 6\}$$

$$S(EG \sim heat) = \{1, 2, 3, 5\}$$
$$S((start \wedge EG \sim heat)) = \{2, 5\}$$

$$S(EF(start \wedge EG \sim heat)) = \{1, 2, 3, 4, 5, 6, 7\}$$

$$S(\sim EF(start \wedge EG \sim heat)) = \emptyset$$
$$S(AG(start \rightarrow AF heat)) = \emptyset$$

Since the first state is not in the set $S(AG(start \rightarrow AF heat))$, the system does not satisfy the specification $AG(start \rightarrow AF heat)$.

Check whether *“Whenever the oven is started, then eventually the oven will heat.”* is satisfied. $AG(start \rightarrow AF\, heat)$

$S(start) = \{2, 5, 6, 7\}$
$S(\sim heat) = \{1, 2, 3, 5, 6\}$

$S(EG \sim heat) = \{1, 2, 3, 5\}$
$S((start \wedge EG \sim heat)) = \{2, 5\}$

$S(EF(start \wedge EG \sim heat)) = \{1, 2, 3, 4, 5, 6, 7\}$

$S(\sim EF(start \wedge EG \sim heat)) = \emptyset$
$S(AG(start \rightarrow AF\, heat)) = \emptyset$

Since the first state is not in the set $S(AG(start \rightarrow AF\, heat))$, the system does not satisfy the specification $AG(start \rightarrow AF\, heat)$.

Check whether *"Whenever the oven is started, then eventually the oven will heat."* is satisfied. $AG(start \rightarrow AF\,heat)$

$S(start) = \{2,5,6,7\}$
$S(\sim heat) = \{1,2,3,5,6\}$

$S(EG \sim heat) = \{1,2,3,5\}$
$S((start \wedge EG \sim heat)) = \{2,5\}$

$S(EF(start \wedge EG \sim heat)) = \{1,2,3,4,5,6,7\}$

$S(\sim EF(start \wedge EG \sim heat)) = \emptyset$
$S(AG(start \rightarrow AF\,heat)) = \emptyset$

Since the first state is not in the set $S(AG(start \rightarrow AF\,heat))$, the system does not satisfy the specification $AG(start \rightarrow AF\,heat)$.

Can you find a counter example to the above formula?

Check whether *“Whenever the oven is started, then eventually the oven will heat.”* is satisfied. $AG(start \rightarrow AF\,heat)$

$S(start) = \{2,5,6,7\}$
$S(\sim heat) = \{1,2,3,5,6\}$

$S(EG \sim heat) = \{1,2,3,5\}$
$S((start \wedge EG \sim heat)) = \{2,5\}$

$S(EF(start \wedge EG \sim heat)) = \{1,2,3,4,5,6,7\}$

$S(\sim EF(start \wedge EG \sim heat)) = \emptyset$
$S(AG(start \rightarrow AF\,heat)) = \emptyset$

Since the first state is not in the set $S(AG(start \rightarrow AF\,heat))$, the system does not satisfy the specification $AG(start \rightarrow AF\,heat)$.

Can you find a counter example to the above formula?

What is the formula to state: *“Whenever the oven is started correctly then eventually it will heat.”*

Check whether *“Whenever the oven is started, then eventually the oven will heat.”* is satisfied. $AG(start \rightarrow AF\,heat)$

$S(start) = \{2, 5, 6, 7\}$
$S(\sim heat) = \{1, 2, 3, 5, 6\}$

$S(EG \sim heat) = \{1, 2, 3, 5\}$
$S((start \wedge EG \sim heat)) = \{2, 5\}$

$S(EF(start \wedge EG \sim heat)) = \{1, 2, 3, 4, 5, 6, 7\}$

$S(\sim EF(start \wedge EG \sim heat)) = \emptyset$
$S(AG(start \rightarrow AF\,heat)) = \emptyset$

Since the first state is not in the set $S(AG(start \rightarrow AF\,heat))$, the system does not satisfy the specification $AG(start \rightarrow AF\,heat)$.

Can you find a counter example to the above formula?

What is the formula to state: *“Whenever the oven is started correctly then eventually it will heat.”* $AG((start \wedge \sim error) \rightarrow AF\,heat)$

A Practical View on Model Checking

Model Checking *mathematically* proves functional properties (specs) on the DUV.

- No tests required - theoretically.
- In practice, testbenches will always have a use during development. Why?

Proving a property is showing that it holds for all possible input combinations, across all execution paths.

- With these benefits comes an obvious drawback:

A Practical View on Model Checking

Model Checking *mathematically* proves functional properties (specs) on the DUV.

- No tests required - theoretically.
- In practice, testbenches will always have a use during development. Why?

Proving a property is showing that it holds for all possible input combinations, across all execution paths.

- With these benefits comes an obvious drawback:

Approach is exponential in complexity wrt size of DUV.

Model Checking: The Practical Side

- Model Checking is typically applied to the **control path,** leaving the datapath for simulation.

Inputs:

- DUV in HDL
- specification/properties (sometimes called rules) to verify
- environment — Why?
  - definition of target system in which DUV will operate

Outputs:

- Documented pass/fail answer (with error trace).

How can FV be integrated into a Sim-Based Verification Flow?

Use FV alongside sim to:

- Check *bus interface protocols* of all modules systematically.
- Check *selected properties of critical complex control blocks* to increase confidence beyond results from testbench.
- Check some small modules exhaustively by model checking not using a testbench at all. ;)
- Let designers write **implementation-level assertions,** not part of verification plan, for checking either formally or in simulation.

The IBM Model Checker: RuleBase

RuleBase [IBM]: symbolic CTL model checking on Verilog/VHDL designs.

Specification language is high-level superstructure over CTL, initially called *Sugar*.

- Deals with industrial-size designs - capacity & robustness.
  - Applied to all microprocessor designs developed in IBM.

Other commercial model checkers:

- **FormalCheck [Cadence]:** Linear time model checking on pre-defined property templates only.
- **Gateprop [Infineon]**
- ...

Practical Experiences with RuleBase

Model Checking the IBM eServer p690:[Source: IBM Power4 Systems. Vol 46, No 1, 2002]

“We applied FFV to some extent on approximately 40 design components throughout the processor and found more than 200 design flaws at various stages and of varying complexity. At least one bug was found by almost every application of FFV.

In most cases, FFV began significantly later than verification.

It is estimated that 15% of these bugs were of extreme complexity and would have been difficult for traditional verification. In some cases, a late bug found in verification or in the laboratory was recreated and its correction verified efficiently with FFV.”

Property Checking: Practical Challenges

Tool Perspective: Capacity

- Need to deal with state-space explosion.

■ For a piece of logic containing n latches, the size of the state machine is 2^n states. Exponential increase in the number of states!

User Perspective: Usability

- How to define *functional properties* (the specification)?
- How to define *assumptions* on input (the environment)?

To overcome the Model Size Problem

Optimizations and heuristics to search through state space:

- BDD-based search

Automated reductions/abstractions:

💡 Reduce size while retaining behaviour!

- cone-of-influence
- flip-flop equivalence

Advanced Model Checking Algorithms:

- on-the-fly checking of *safety* formulas
  - verification takes place together with construction of state space
- bounded model checking
- guided search

Usability: How to write properties?

- **PSL/Sugar language:** [April 22, 2002] Accellera selects Sugar 2.0 from IBM as basis of IEEE standard specification language. [June 2, 2003]
- Accellera approves four new standards for language-based design verification, including Property Specification Language (PSL) based on Sugar 2.0.

”A change is taking place in the way we design and verify our designs that will revolutionize the industry and result in the equivalent of a synthesis productivity breakthrough in verification. This change demands that we move from natural language forms of specification to forms that are mathematically precise and verifiable, and lend themselves to automation. The PSL 1.01 standard offers an *opportunity to enable this huge leap in productivity of specification, design, and verification,*” said Harry Foster, Accellera Formal Verification Technical Committee Chair.

PSL/Sugar

- Developed to address shortcomings of natural language forms of specification.
- Provides **designers** with standard means of specifying design properties using a *concise syntax with clearly defined formal semantics.*
- Allows **RTL implementer** to *capture design intent* in a verifiable form.
- Enables **verification engineers** to validate that the implementation satisfies its specification with dynamic (that is, simulation) and static (that is, formal) verification.

Technical Features of PSL/Sugar

Engineer-friendly syntax and semantics:

- easy specification of overlapping sequences and events
- textual equivalent of timing diagrams
- extensive syntactic sugaring

Fits many paradigms of ABV:

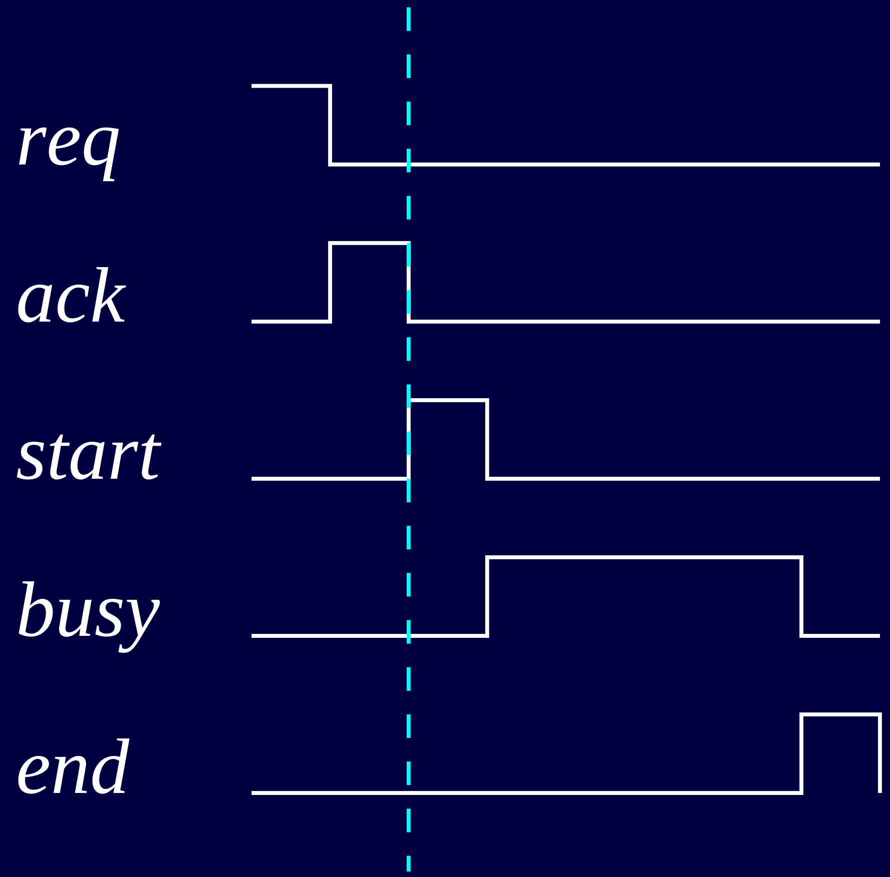
- linear language foundation for simulation and LTL-based model checking
- optional branching time extension for CTL-based model checking

Multiple HDL flavours: Verilog, VHDL, EDL

Support for multiple clocks and asynchronous design

PSL/Sugar Example

{ req; ack }|=>{ start; busy[*];end }



Example PSL/Sugar Property

"Whenever we see an assertion of signal req, followed by an assertion of signal ack, which is not followed by an assertion of signal cancel, we should see - starting at the next cycle - an assertion of signal start_trans, followed by between 1 and 8, not necessarily consecutive assertions of signal data_valid, followed by an assertion of signal write_end."

```
always { req; ack; !cancel } |=>
{ start_trans; data_valid [=1..8]; write_end }
```

Natural and concise way for formulating DUV properties.

++ *Provides a standard means for hardware designers and verification engineers to rigorously document the design specification.*

Defining Assumptions on Input

Syntax:

- Same as for properties or alternatively FSM syntax (HDL or IBM EDL).

Why do we need “environmental assumptions”?

Defining Assumptions on Input

Syntax:

- Same as for properties or alternatively FSM syntax (HDL or IBM EDL).

Why do we need “environmental assumptions”?

- Avoid “False Negatives”
- Only consider legal inputs to the DUV.

Under illegal inputs, artificial (non-realistic) failures can be reported!

- Limit search space:
  - Reduce set of applicable input behaviours.

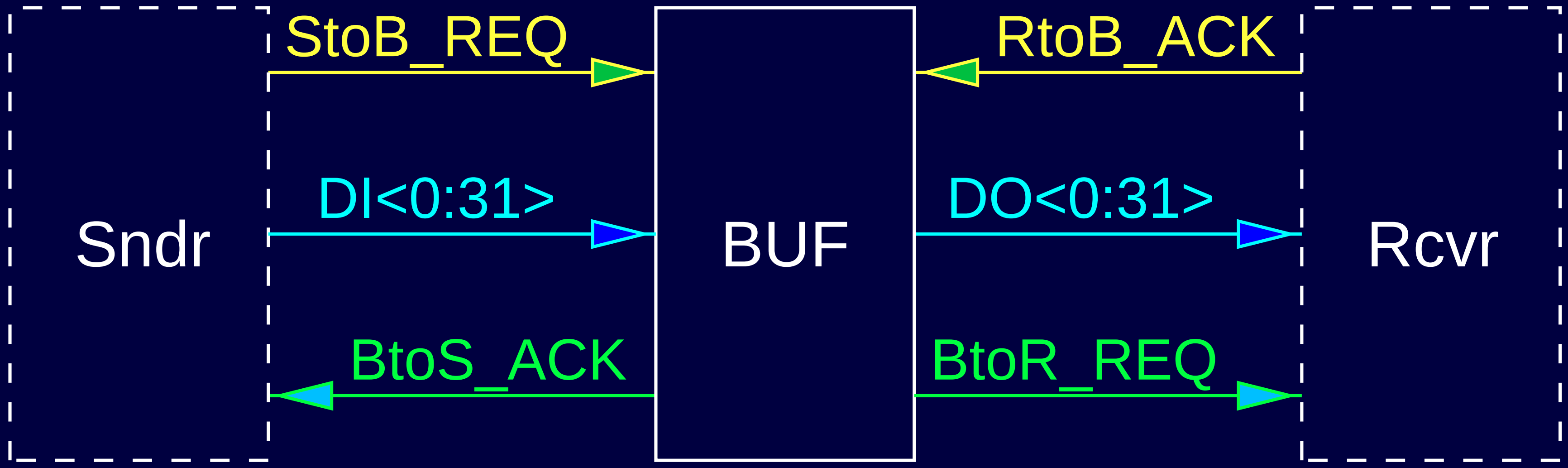
  $\Rightarrow$ Reduce state-space size!

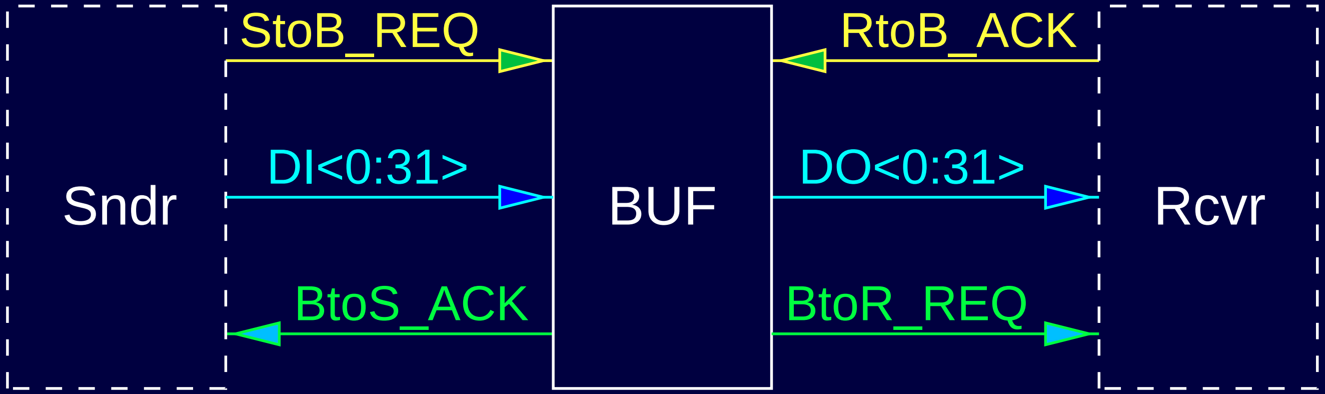
Environment: Needs to exhibit faithful representation of reality.

- Needs to be designed so it can be flexibly used to restrict search.

Example: Another Buffer

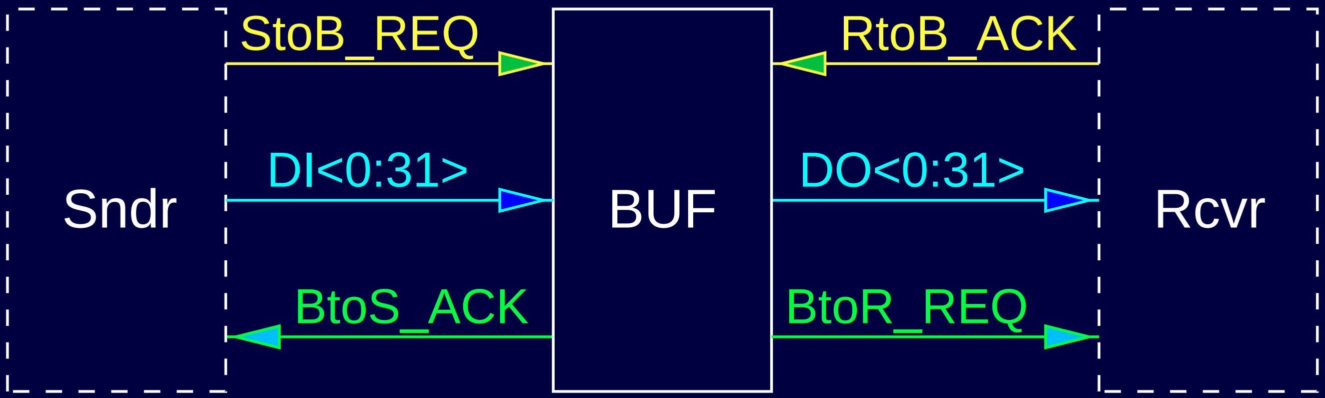
BUF is a design block that buffers a word of data (32 bits) sent by a sender to a receiver. It has two control inputs, two control outputs, and a data bus on each side:





Communication (on both sides) takes place by means of a 4-phase handshaking protocol as follows:

When the sender has data to send to the receiver, it initiates a transfer by putting the data on the data bus and asserting StoB_REQ (Sender to Buffer REQuest). If BUF is free, it reads the data and asserts BtoS_ACK (Buffer to Sender ACKnowledge). Otherwise the sender waits. After seeing BtoS_ACK, the sender may release the data bus and deassert StoB_REQ. To conclude the transaction, BUF deasserts BtoS_ACK.



Communication (on both sides) takes place by means of a 4-phase handshaking protocol as follows:

When BUF has data, it initiates a transfer to the receiver by putting the data on the data bus and asserting BtoR_REQ (Buffer to Receiver REQuest). If the receiver is ready, it reads the data and asserts RtoB_ACK (Receiver to Buffer ACKnowledge). Otherwise, BUF waits. After seeing RtoB_ACK, BUF may release the data bus and deassert BtoR_REQ. To conclude the transaction, the receiver deasserts RtoB_ACK.

Property Specification

- *Basic functional properties*
- *Protocol properties*
- *Data-path properties*

The next few examples will also expose you to Sugar 2.0 syntax. :)

Basic Functional Properties

1. Overflow (two writes without a read in between) cannot occur.
2. Underflow (two reads without a write in between) cannot occur.

```
rule ack_interleaving {

  formula
    "No overflow: RtoB_ACK is asserted between any two BtoS_ACK assertions "
    {
    always
    { [*] ; !RST & rose(BtoS_ACK) ; true }( rose(RtoB_ACK) before rose(BtoS_ACK) )
    }

  formula
  "No underflow: BtoS_ACK is asserted between any two RtoB_ACK assertions"
  {
  always
  { [*] ; !RST & rose(RtoB_ACK) ; true }( rose(BtoS_ACK) before rose(RtoB_ACK) )
  }
}
```

Handshaking Properties

1. The protocol sequence must always be honoured

2. Focus on Request signals first
   – Request signals will not be prematurely asserted
   – Once asserted, Request signals will stay high until acknowledged

3. Next, focus on Acknowledge signals
   – Acknowledge signals cannot spontaneously be asserted
   – Once asserted, an Acknowledge signal will stay high until the triggering Request signal is turned off

Handshaking Properties: LHS (Sender)

```
rule four_phase_handshake_left

{

  formula   "A request can not be raised when ack is high "
  {  always {  [*] ; !StoB_REQ & BtoS_ACK ; StoB_REQ } (false)

  formula   "A request can not be lowered when ack is low"
  {  always {  [*] ; StoB_REQ & !BtoS_ACK ; !StoB_REQ } (false)

  formula   "An acknowledge can not be raised when req is low"
  {  always {  [*] ; !BtoS_ACK & !StoB_REQ ; BtoS_ACK } (false)

  formula   "An acknowledge can not be lowered when req is high"
  {  always {  [*] ; BtoS_ACK & StoB_REQ ; !BtoS_ACK } (false)

}
```

Handshaking Properties: RHS (Receiver)

```
rule four_phase_handshake_right

{

  formula   "A request can not be raised when ack is high"
  {  always {  [*] ; !BtoR_REQ & RtoB_ACK ; BtoR_REQ } (false)  }

  formula    "A request can not be lowered when ack is low"
  {  always {  [*] ; BtoR_REQ & !RtoB_ACK ; !BtoR_REQ } (false) }

  formula    "An acknowledge can not be raised when req is low"
  {  always {  [*] ; !RtoB_ACK & !BtoR_REQ ; RtoB_ACK } (false) }

  formula   "An acknowledge can not be lowered when req is high"
  {  always {  [*] ; RtoB_ACK & BtoR_REQ ; !RtoB_ACK } (false)  }

}
```

Data Path Properties

Basic transfer property:

Data sent to receiver is the same as received from sender.

To specify this in Sugar, we can use **`next_event (b)(f)`**

- The next time that **`b`** holds, **`f`** must hold.

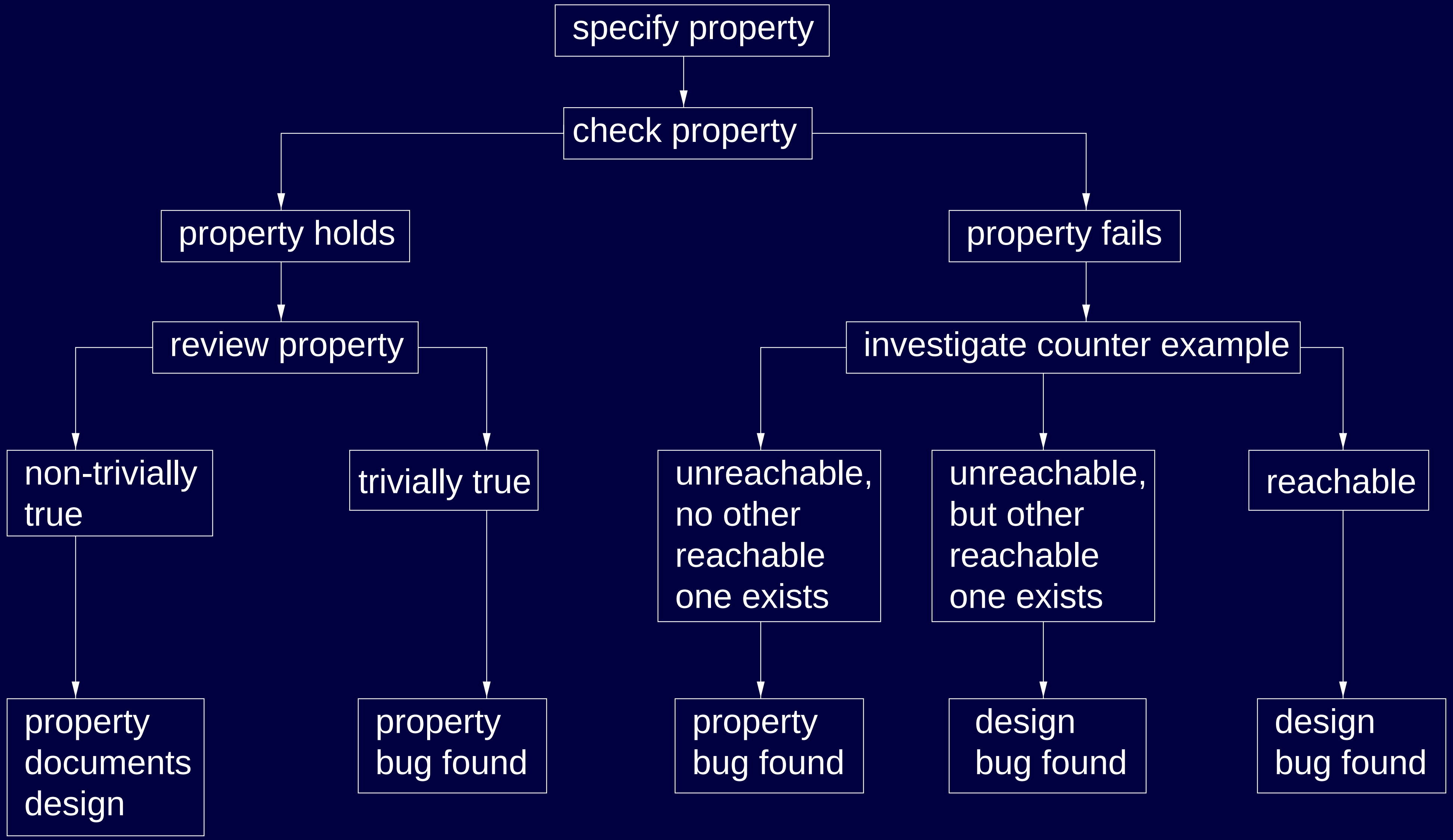
A conceptual extension of the basic next operator!

```
rule checking_data{

    formula
    { always  ((!RST & rose(BtoS_ACK) &  DI(0))  -> next_event(rose(RtoB_ACK))( DO(0)) )
    }

    formula
    { always  ((!RST & rose(BtoS_ACK) &  !DI(0))  ->  next_event(rose(RtoB_ACK)) ( !DO(0)) )
    }
}
```

Property Checking Strategy - in practice



[Credits: Decision diagram adopted from Infineon Technologies Training Material.]

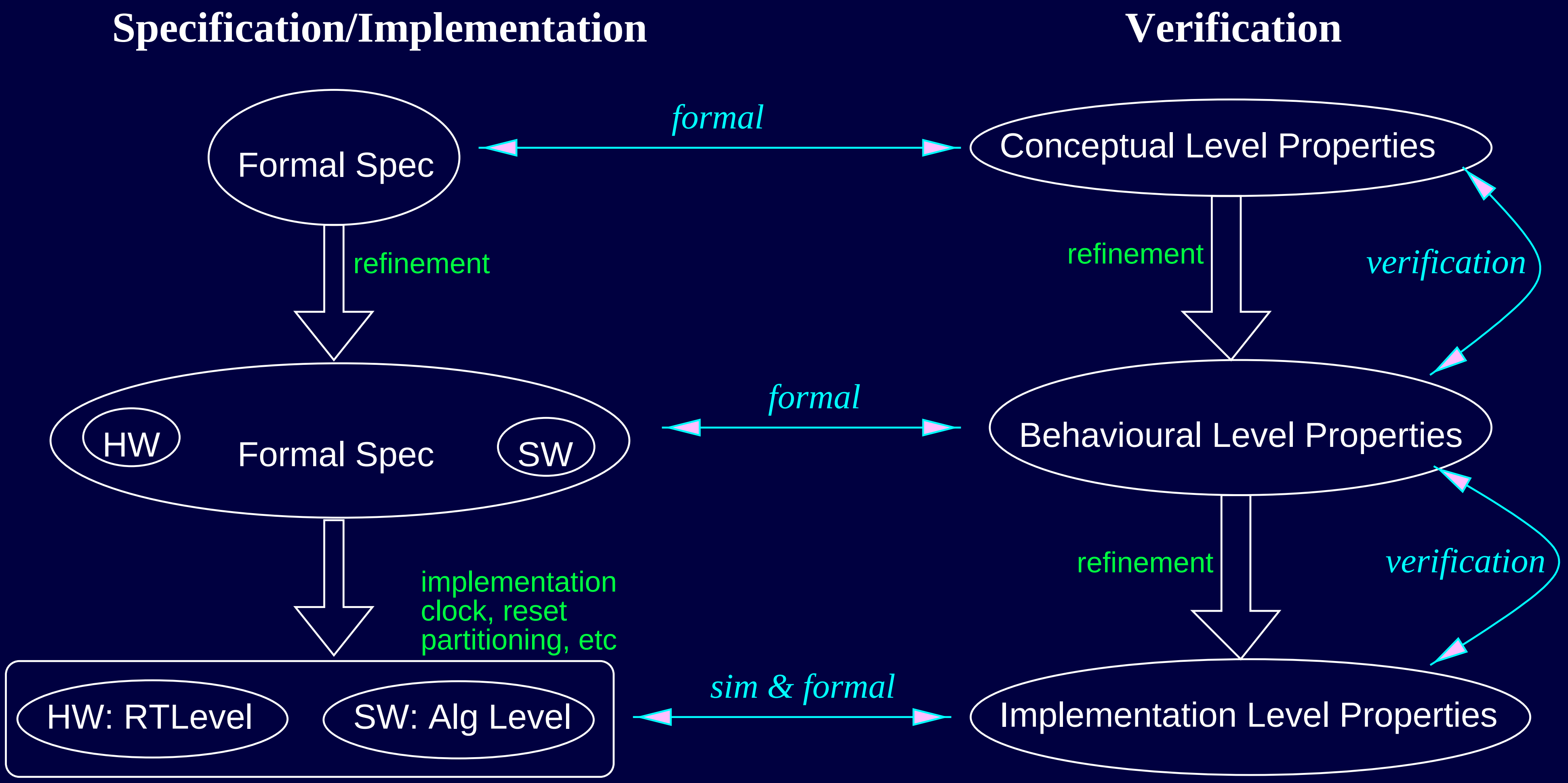
Property Checking Methodology to Overcome Size Limitations

- Verify **control logic** alone, leave datapath for simulation
- Abstract out internal parts = structural abstraction
  - ■ **black-boxing** (Show example on black board!)
    - Might make debug difficult. WHY?
- Degenerate design parts by **limiting environmental behaviour**

- **Partitioning:**
  - ++ Divide into units of moderate/manageable size.
  - – *Best done early: Design for Verification!*
  - – – Verify local properties - often not well documented/specified!

Plan partitions in advance, identify properties, document these well.

The Future: Specification/Design for Verification



Property-oriented Development/Refinement Flow!

Combining Verification Techniques

Combining Model Checking with Simulation:

- To identify illegal coverage points. [functional coverage]

Prevents automatic coverage closure algorithms from getting lost in loops.

- To monitor and direct simulation.

■ Translate sim steps to protocol state transitions using a refinement mapping.

Verify transitions against protocol spec using model checker.

■ Translate paths through FSM into sim stimulus. [CDG]

Clever stimulus generation for simulation to close coverage.

Very hot topic of research and development.

Summary: Functional Formal Verification

Model Checking is a practical Formal Verification Method

- fully automatic
- produces counter examples

Property verification via Model Checking is powerful.

Start from beginning:

- Design for Verification!
- Pragmatically approach your verification problem.
- Make a thorough verification plan.