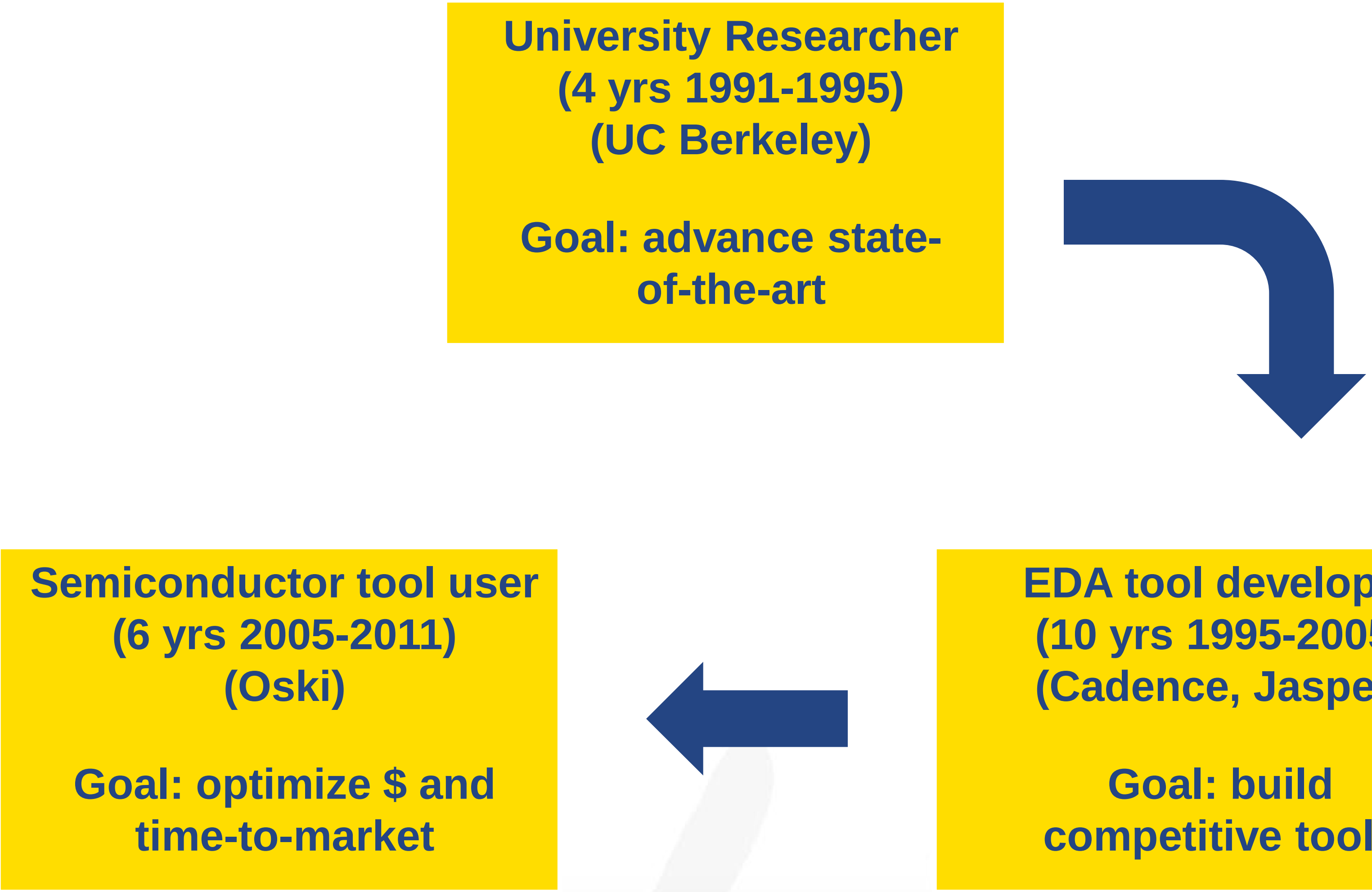
Deploying formal in a simulation world

VIGYAN SINGHAL
OSKI TECHNOLOGY

Formal from different vantage points



University Researcher
(4 yrs 1991-1995)
(UC Berkeley)

Goal: advance state-of-the-art

EDA tool developer
(10 yrs 1995-2005)
(Cadence, Jasper)

Goal: build competitive tools

Semiconductor tool user
(6 yrs 2005-2011)
(Oski)

Goal: optimize $ and time-to-market

Formal in academia

- Goal: advance state-of-the-art
- Areas of concern
  - Temporal logics (CTL, CTL*, PLTL)
  - Fairness and ω-automata
  - Complexity
    - Known: CTL Model checking linear time complexity (size of FSM)
    - Proved: CTL model checking PSPACE-complete (size of design)
- Time to returns: almost infinite
- Anecdote: “model check a 9-state FSM at Motorola”

Formal in EDA company

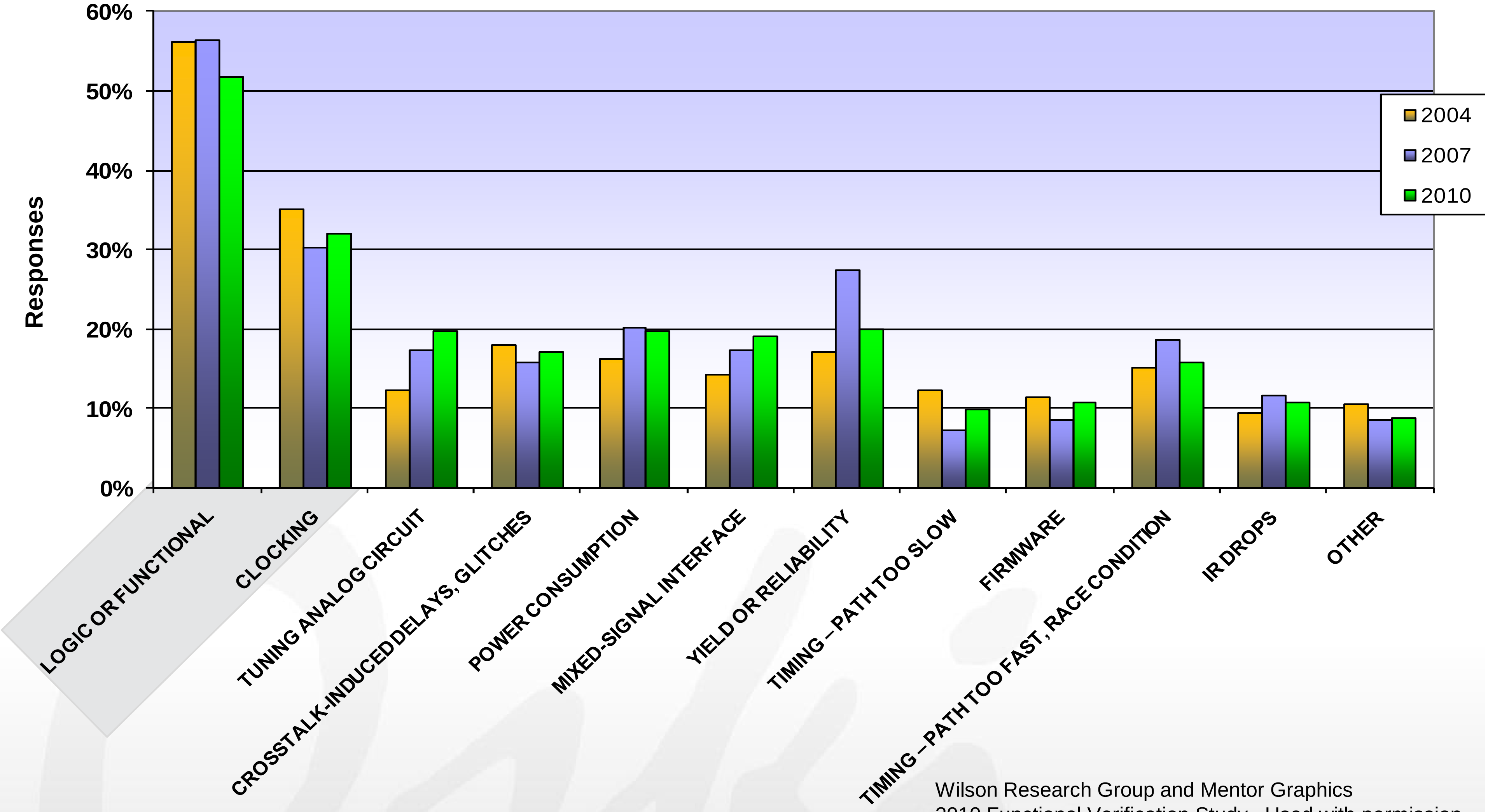
- Goal: build competitive tools
- Areas of concern
  - Verilog/SystemVerilog parsing
  - User interface and GUI
  - Property synthesis: PSL and SVA
- Time to returns: 4-5 years
- Anecdote: “lost an eval at Intel because tool ran for 28 days”

Formal in semiconductor industry

- Goal: optimize $ and time-to-market
- Areas of concern
  - Verification planning
  - Metrics to measure progress, and when we are done
  - Integrate simulation and formal planning and results
  - Abstraction (and reductions) are key to making formal productive
- Time to returns: 3-6 months
- Anecdote: “how did you miss a bug on a formally verified block?”

Types of post-silicon flaws

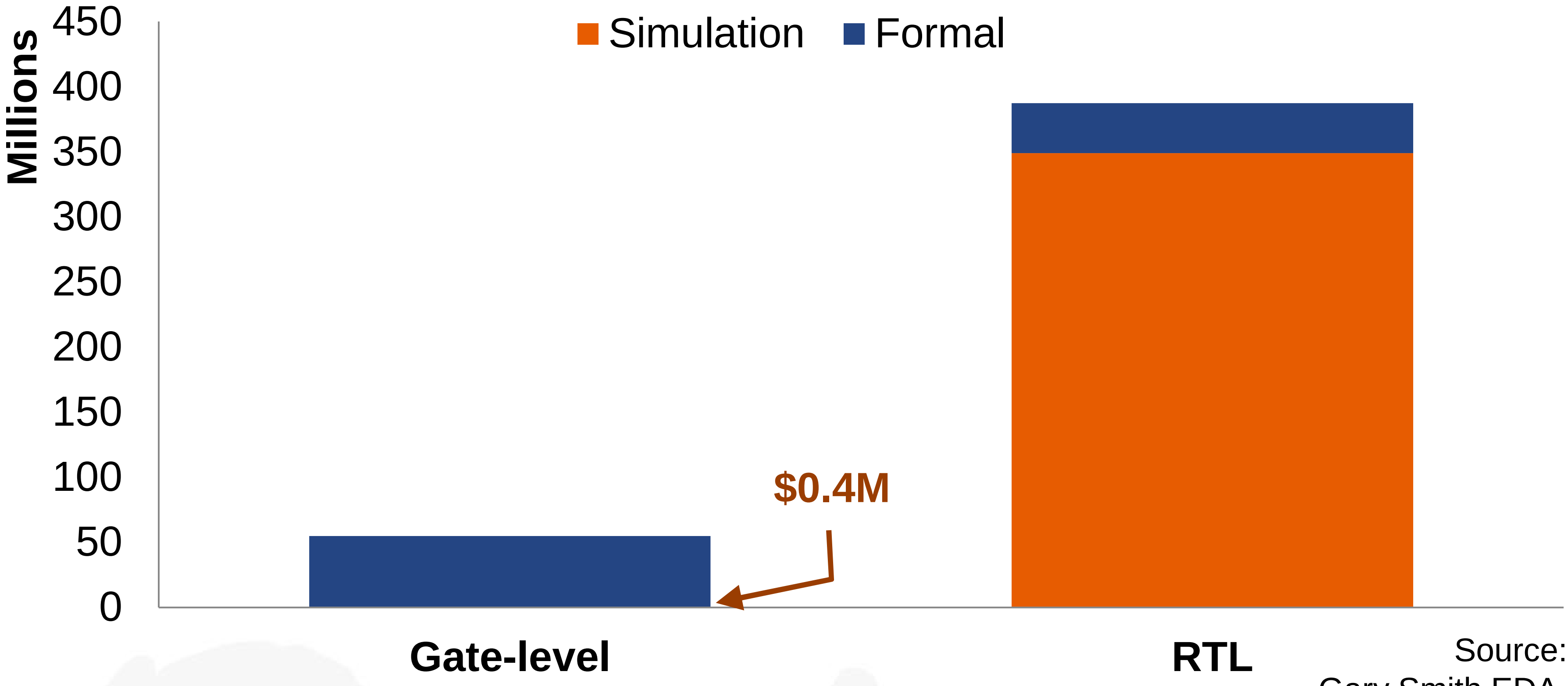
Verification is the still the largest problem



Wilson Research Group and Mentor Graphics
2010 Functional Verification Study. Used with permission.

*Verification market size (2009)**

* excluding analog



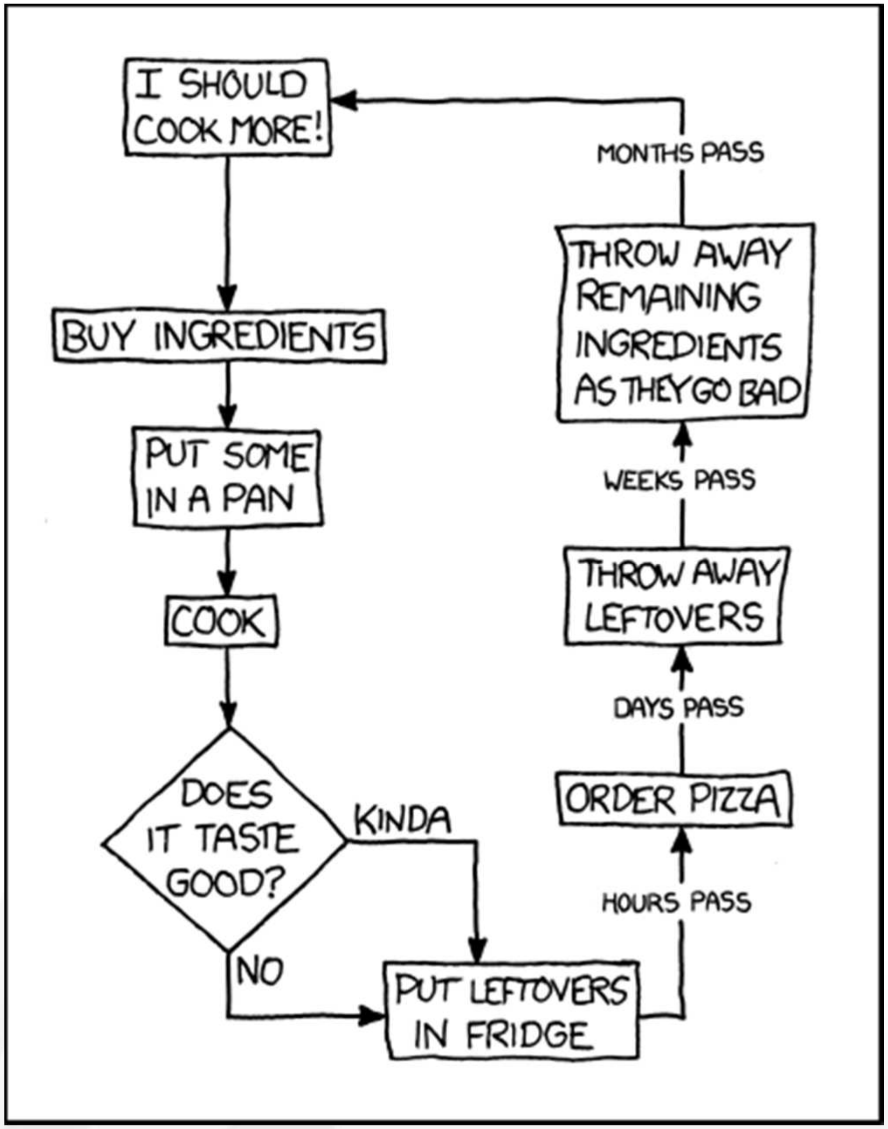
Source:
Gary Smith EDA,
October 2010

- **Gate-level formal (equivalence checking)**
  - **Then (1993):** Chrysalis; **Now:** Cadence (Verplex), Synopsys
- **RTL formal (model checking)**
  - **Then (1994):** Averant, IBM; **Now:** Jasper, Mentor (0-In)

Formal tool usage in industry

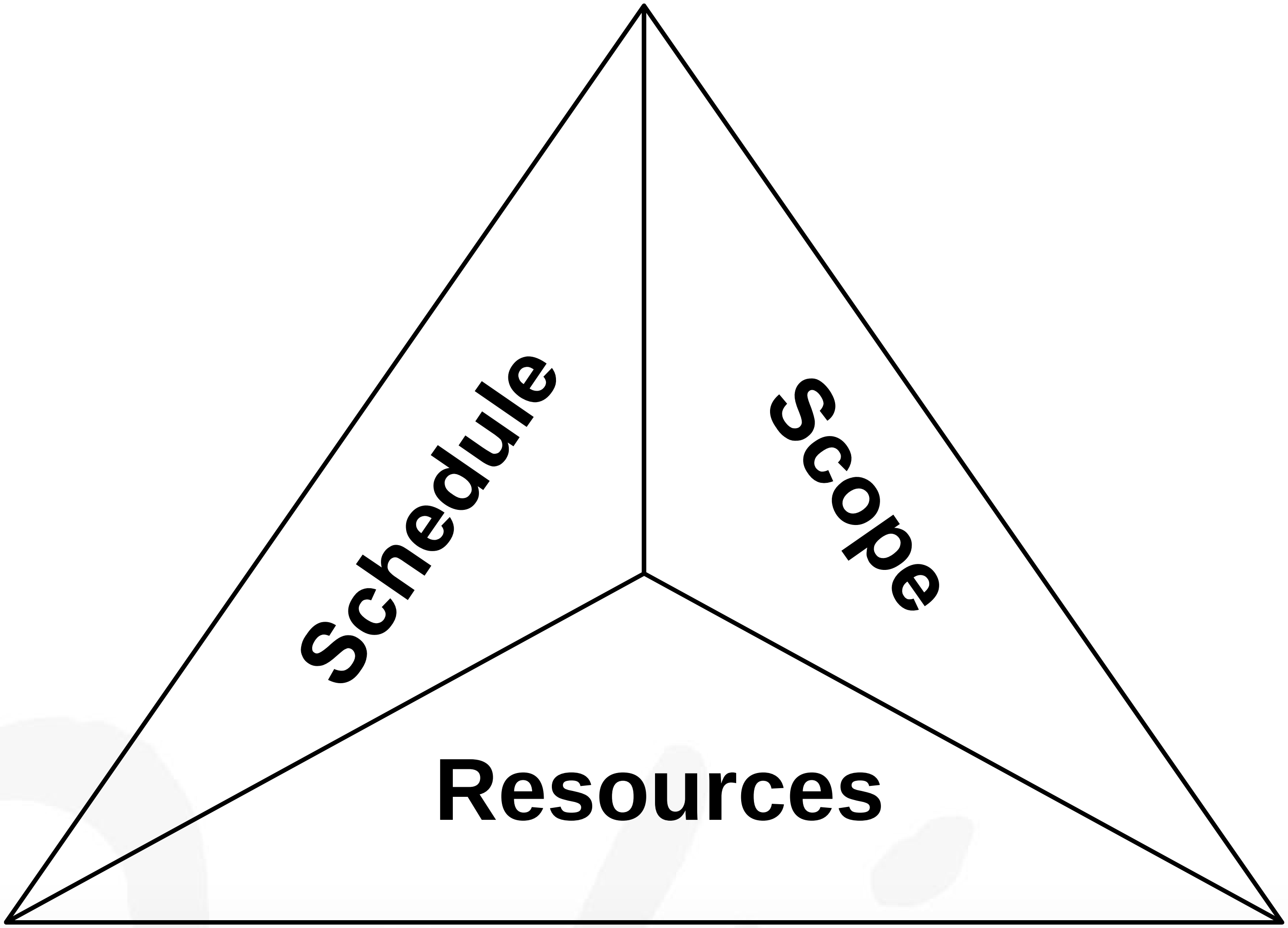
- Around for 20 years
- Expectations has been set high
  - Low efforts for constraints
  - Tools run fast enough
- Expectations have been set low
  - Only verify local assertions
  - No End-to-End proofs
- Perception: low !/$

- Training and staffing
  - Few places to learn formal application
  - Single user should not do both formal and simulation



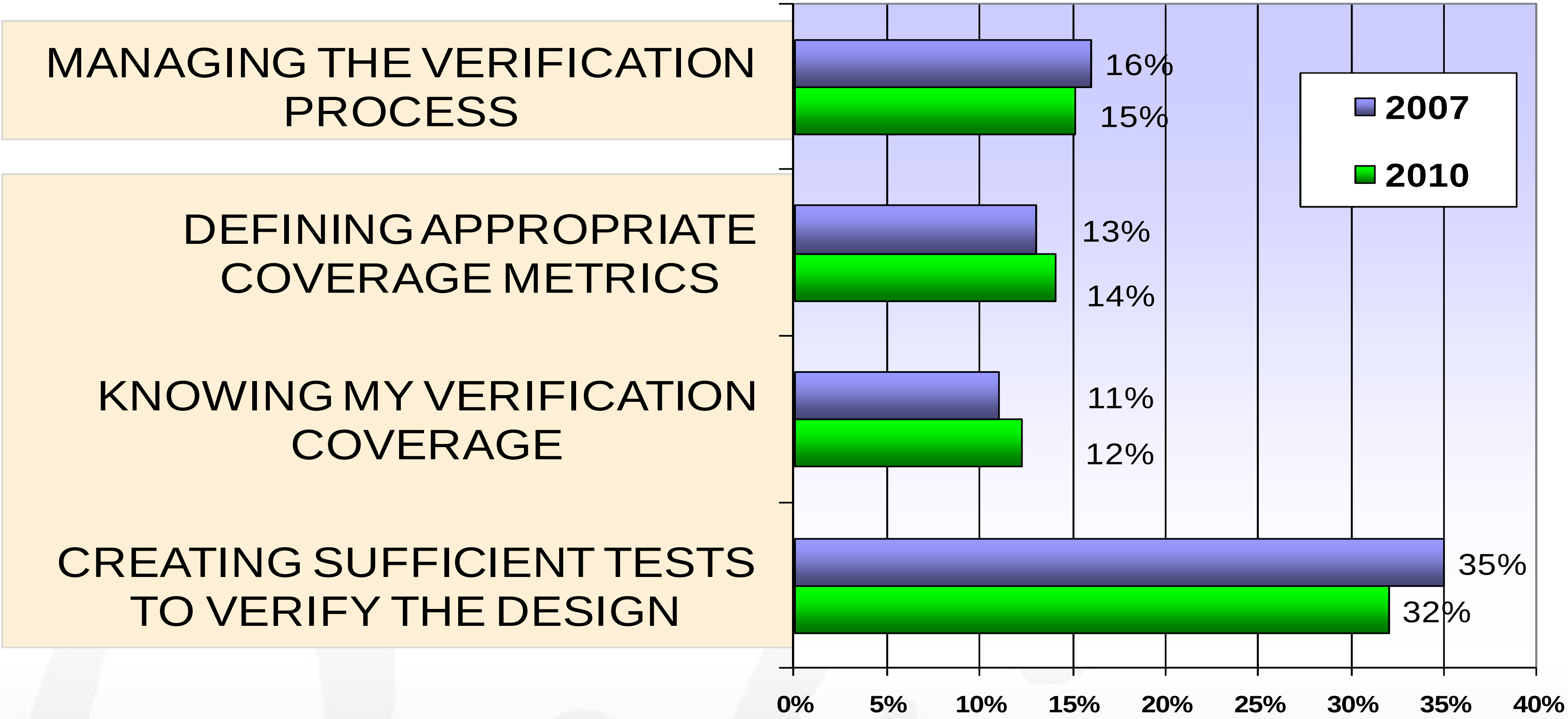
Source: xkcd.com

Tradeoffs in design flow



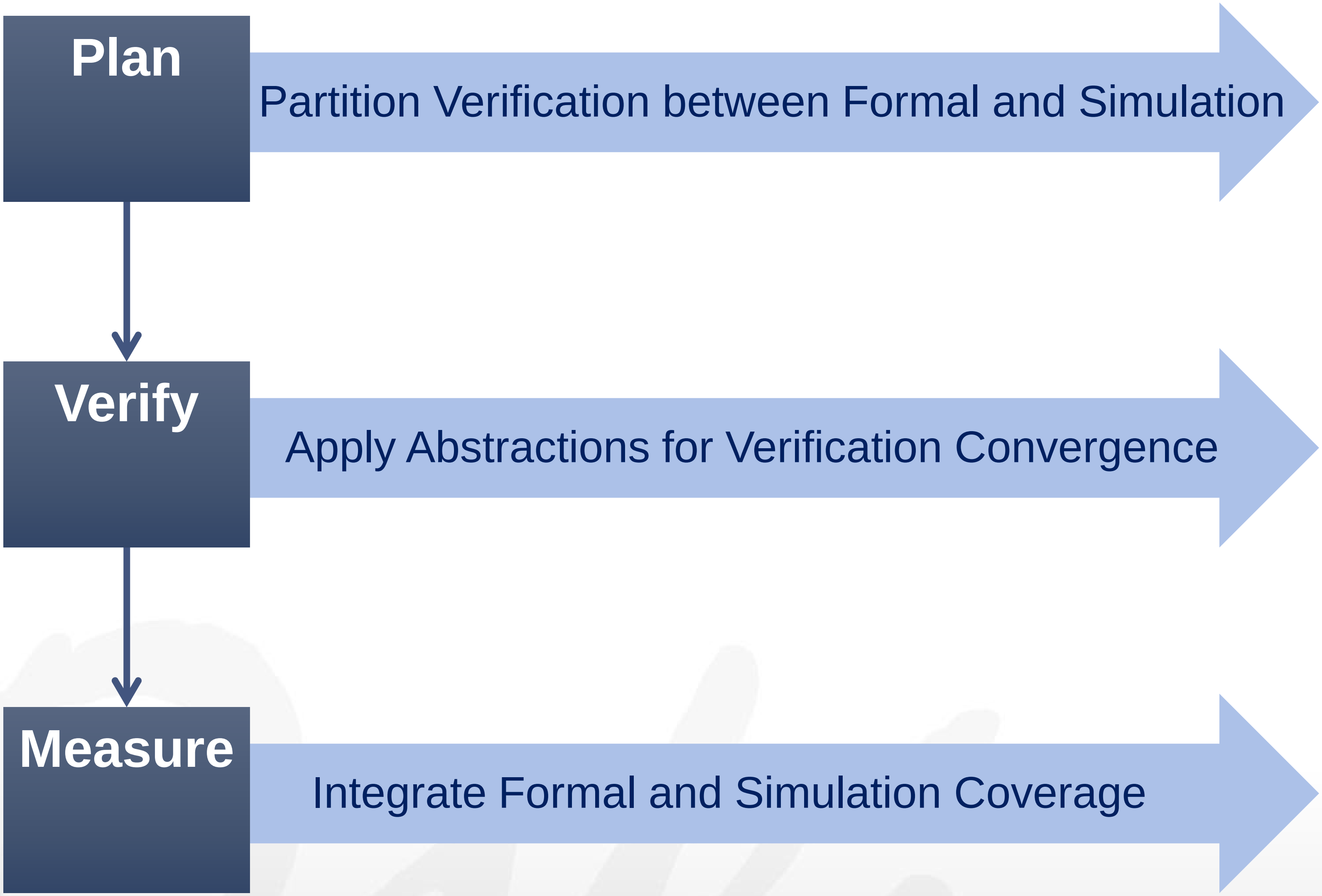
Biggest challenges needing solutions

- Verification management and coverage



Wilson Research Group and Mentor Graphics
2010 Functional Verification Study, Used with permission.

Achieving verification closure



Plan

Partition Verification between Formal and Simulation

Verify

Apply Abstractions for Verification Convergence

Measure

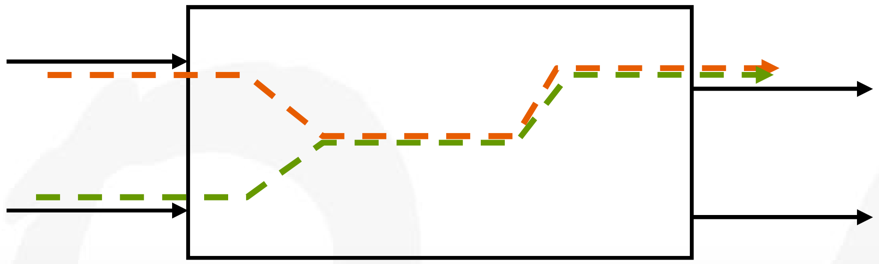
Integrate Formal and Simulation Coverage

Where to apply model checking

“Control”, “Data Transport” designs

- Arbiters of many kinds
- Interrupt controller
- Power management unit
- Credit manager block
- Tag generator
- Schedulers

- Bus bridge
- Memory Controller
- DMA controller
- Host bus interface
- Standard interfaces (PCI Express, USB)
- Clock disable unit



Multiple, concurrent streams

Hard to completely verify using simulation

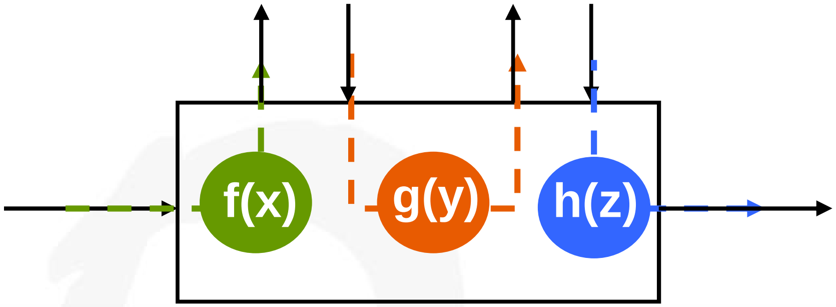
“10 bugs per 1000 gates”

-Ted Scardamalia, IBM

Where not to apply model checking

“Data transform” designs

- Floating point unit
- Graphics shading unit
- Inverse quantization
- Convolution unit in a DSP chip
- MPEG decoder
- Classification search algorithm
- Instruction decode



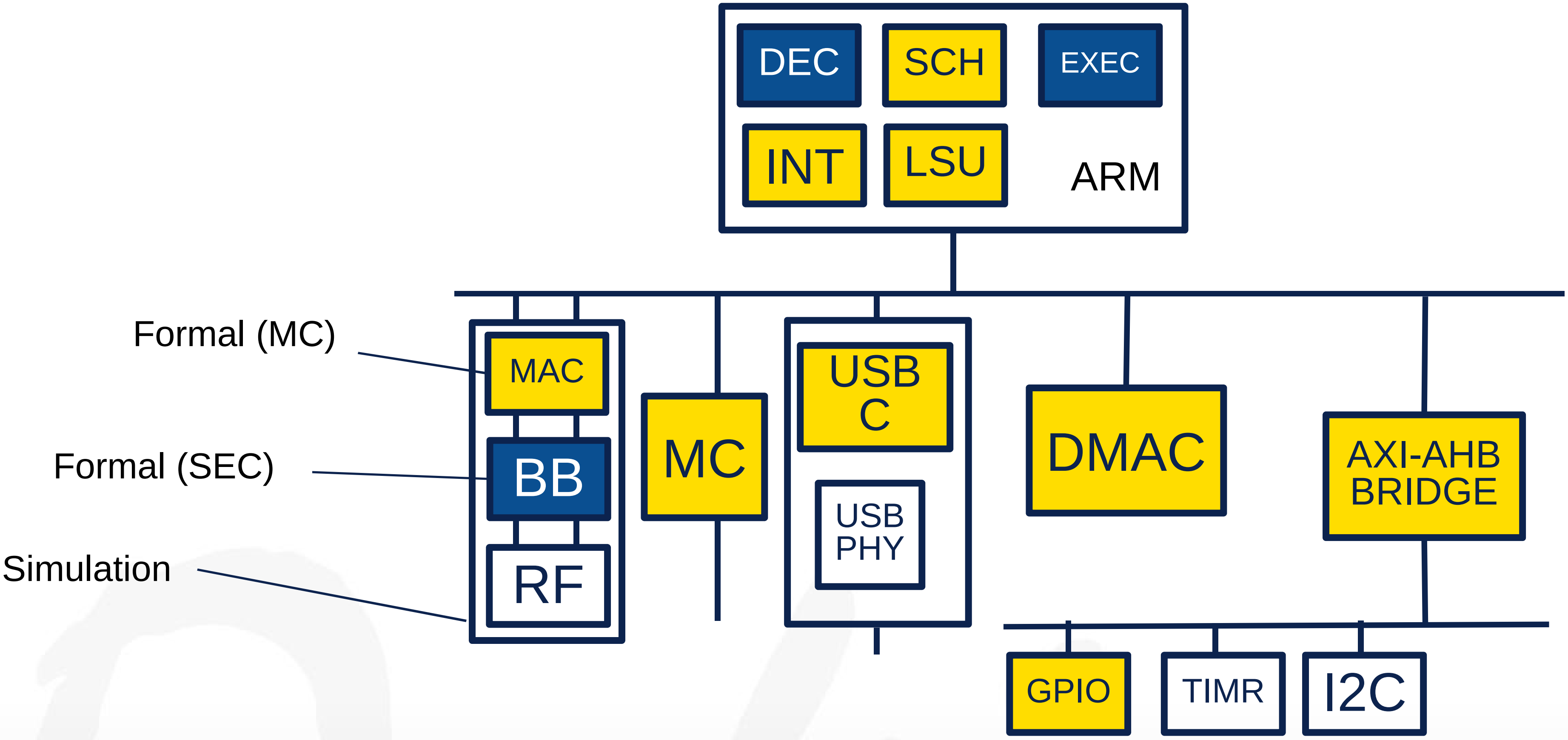
Single, sequential functional streams

“2 bugs per 1000 gates”

-Ted Scardamalia, IBM

Formal (MC, SEC) and simulation strengths*

* SEC = Sequential Equivalence Checking (RTL vs C model)



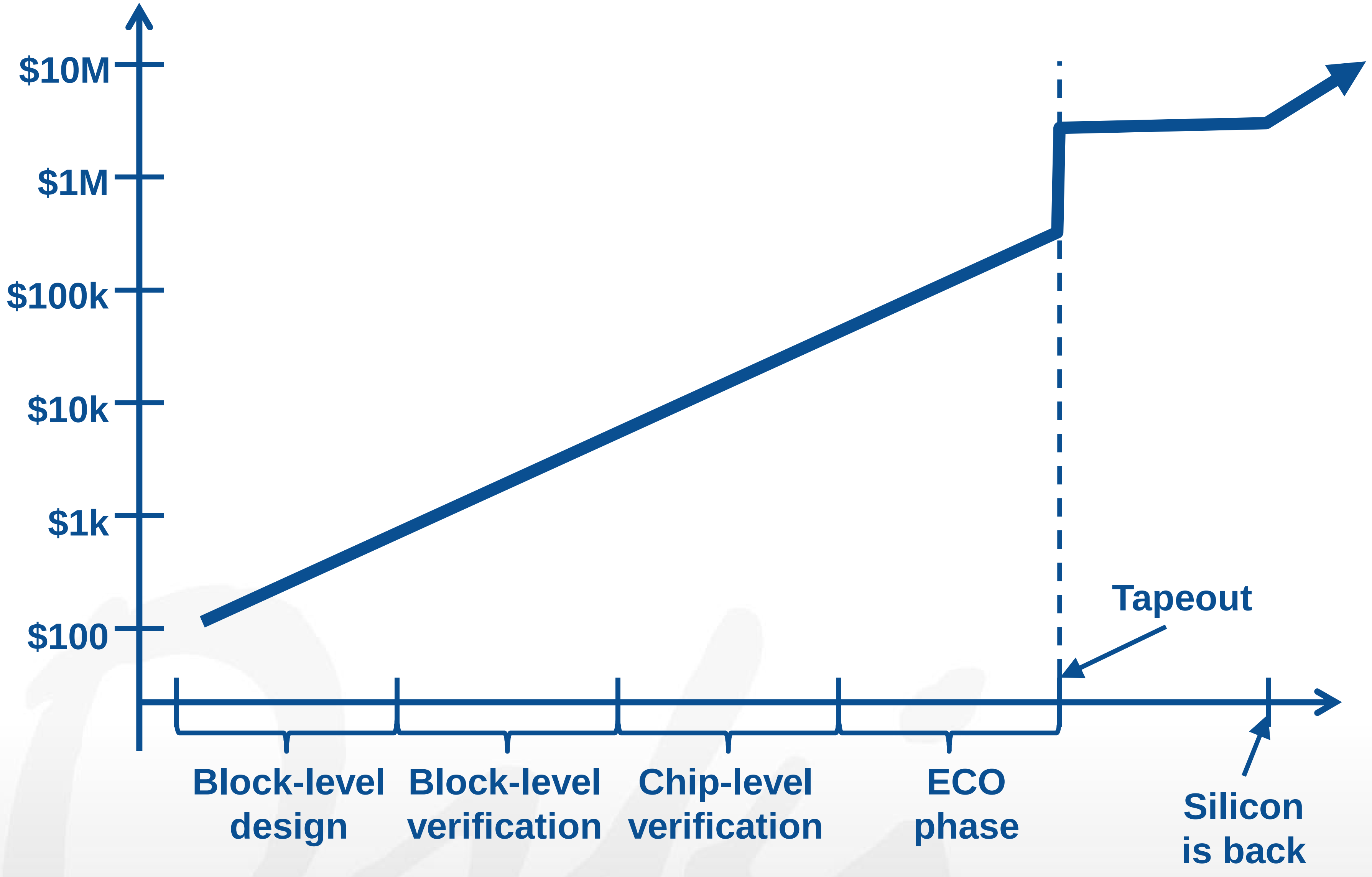
How perfect does formal have to be?

Graphic: MacGregor Marketing

- Not all bugs need to found/fixed
- Formal does not need to find the last bug
- Usually bounded proofs are good enough

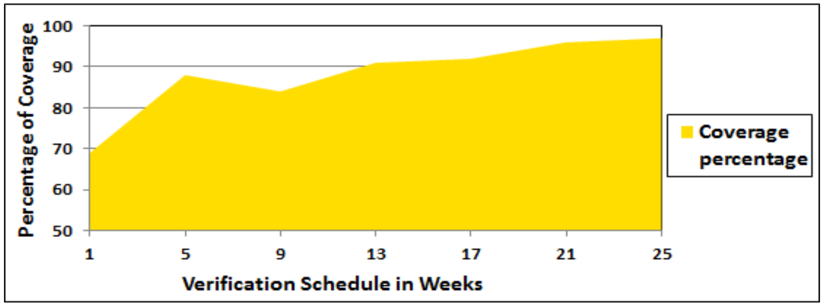
  (if bound is good enough!)
- Formal has to be more cost-effective than the alternative

Bug-fix cost rises exponentially

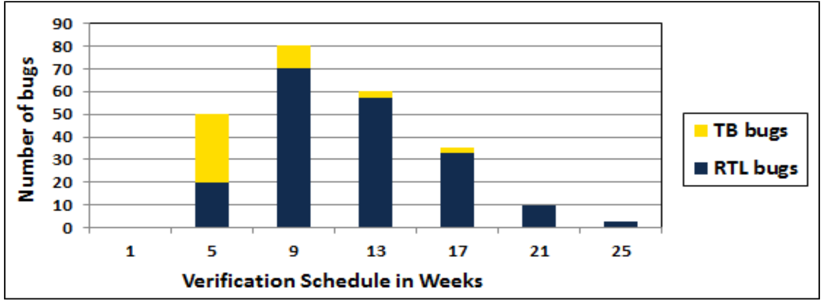


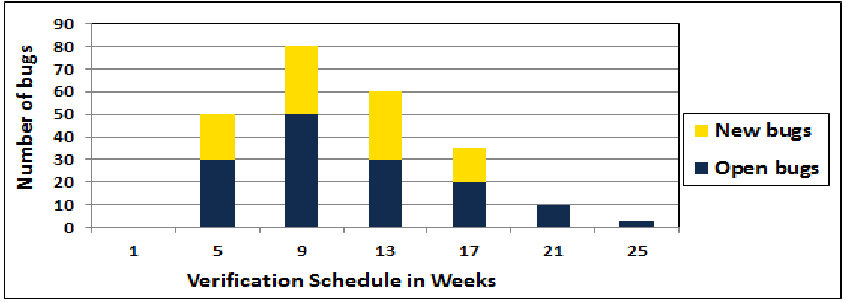
Verification manager's dashboard

Coverage tracking

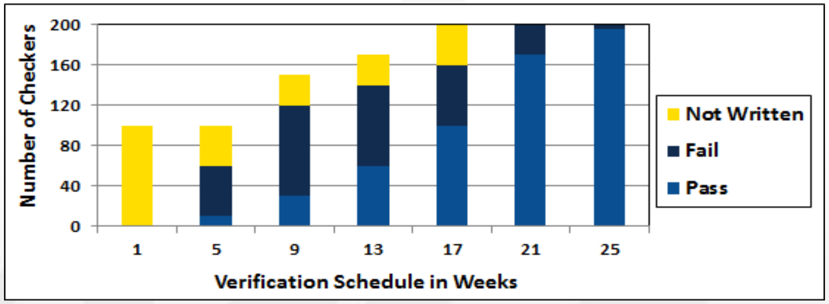


Bug tracking

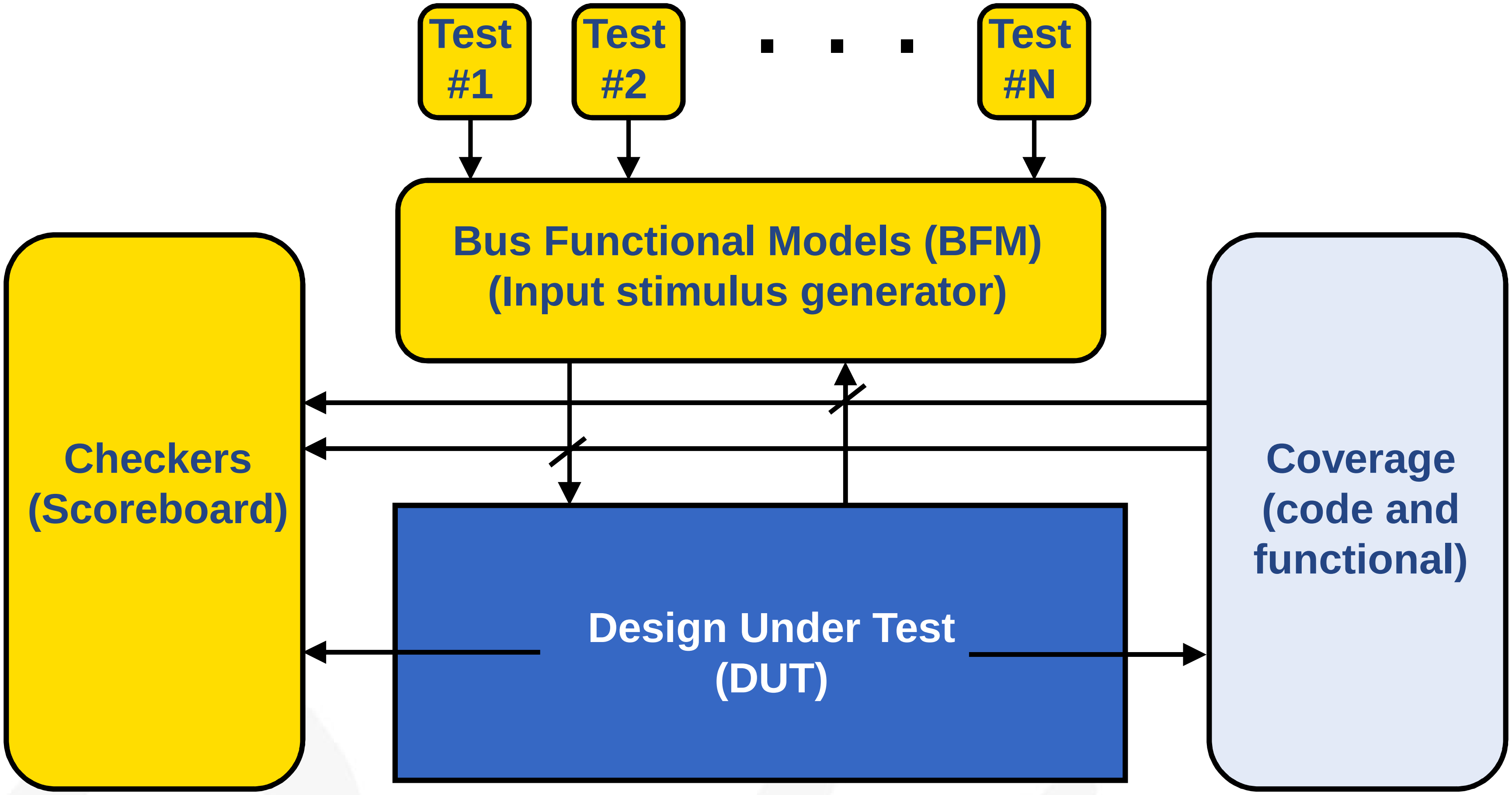




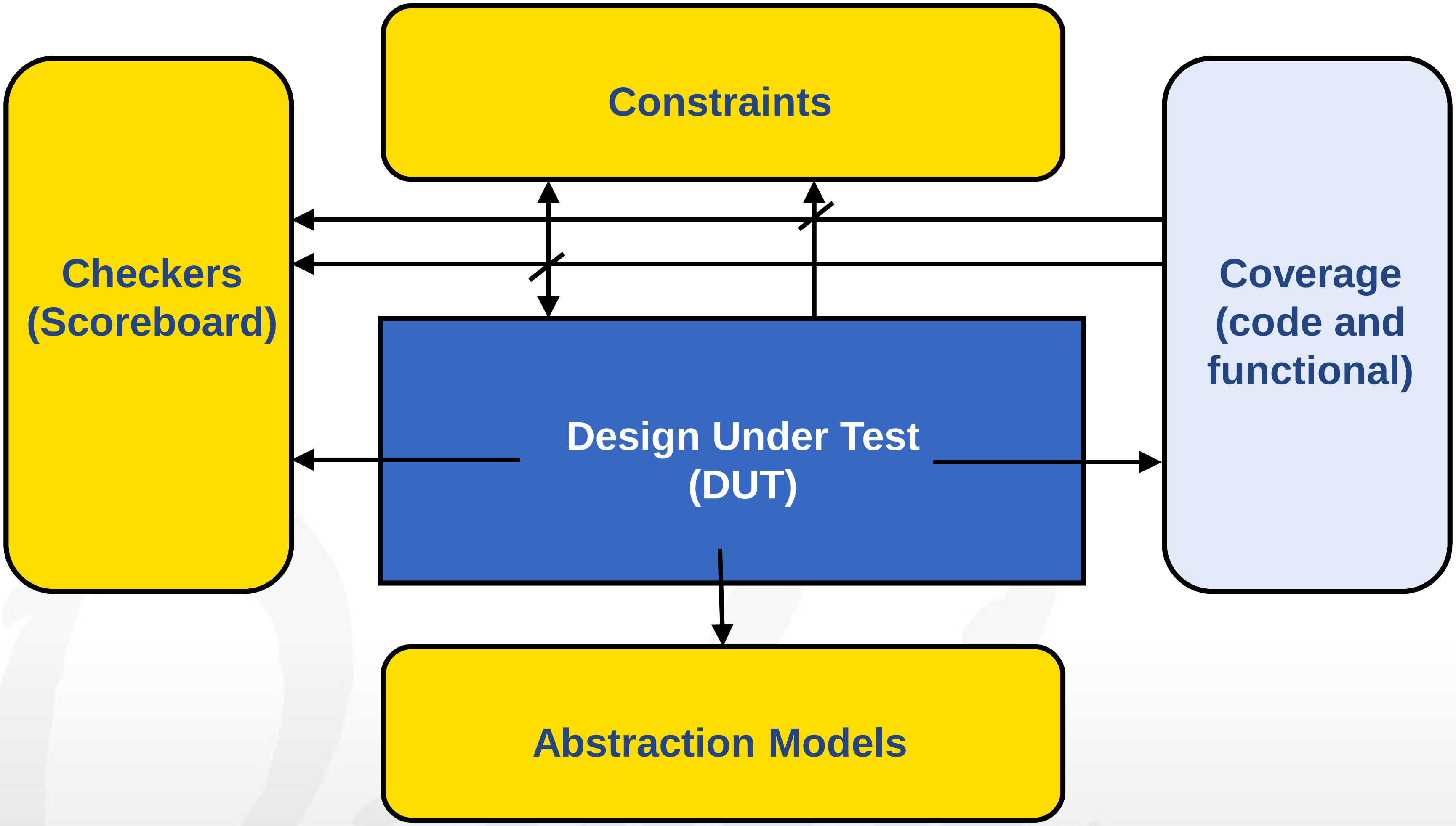
Runtime status



A simulation testbench



A formal testbench



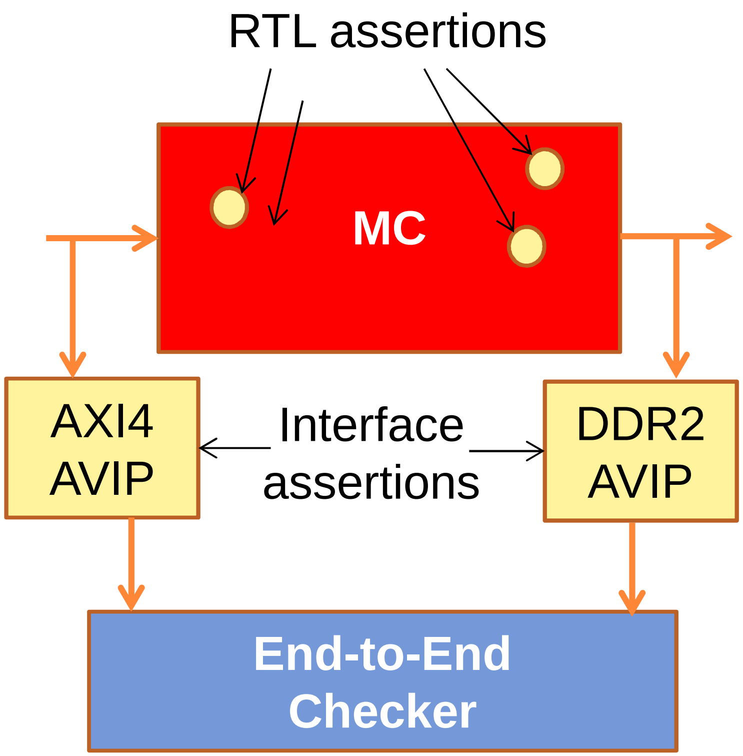
Three Cs of Formal

- Checkers
- Constraints
- Complexity
  - (using Abstraction Models)

- … and Coverage (to measure completeness of formal)

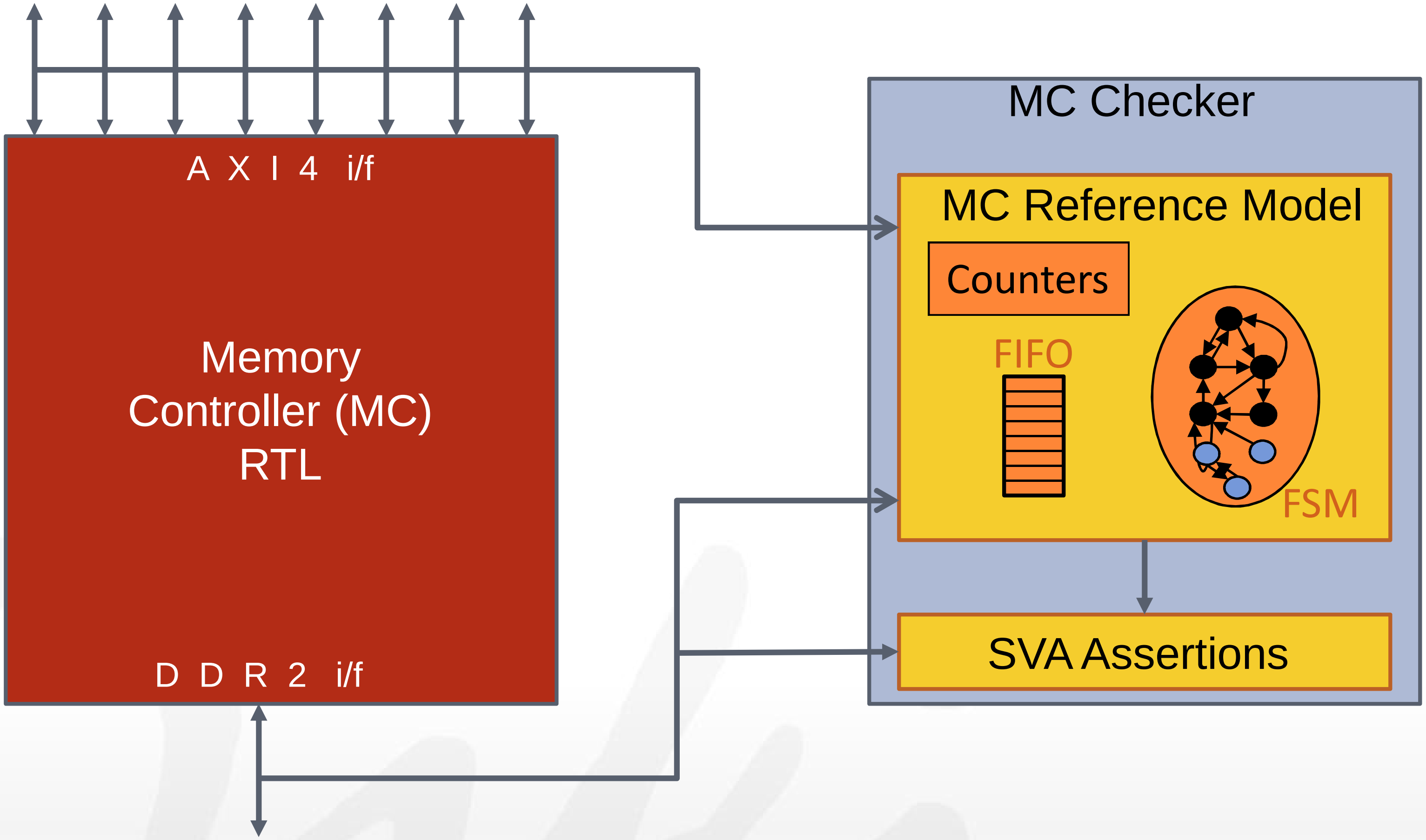
Traditional formal verification

- Usually based on Local Checkers:
  1. RTL assertions
  2. Interface assertions
- Useful for bug hunting
  - Not for finding all/most bugs, or as replacement for simulation effort
- For replacing simulation, need End-to-End Checkers
- Run into complexity barrier
  - For medium- or large-sized designs, run into state space explosion
  - Without Abstraction Models, cannot scale complexity



Checkers (End-to-End)

- For End-to-End formal verification, less than 5% of Checker code is SVA; rest is SV or Verilog
  - (Synthesizable) Reference model is typically as big an effort as the RTL



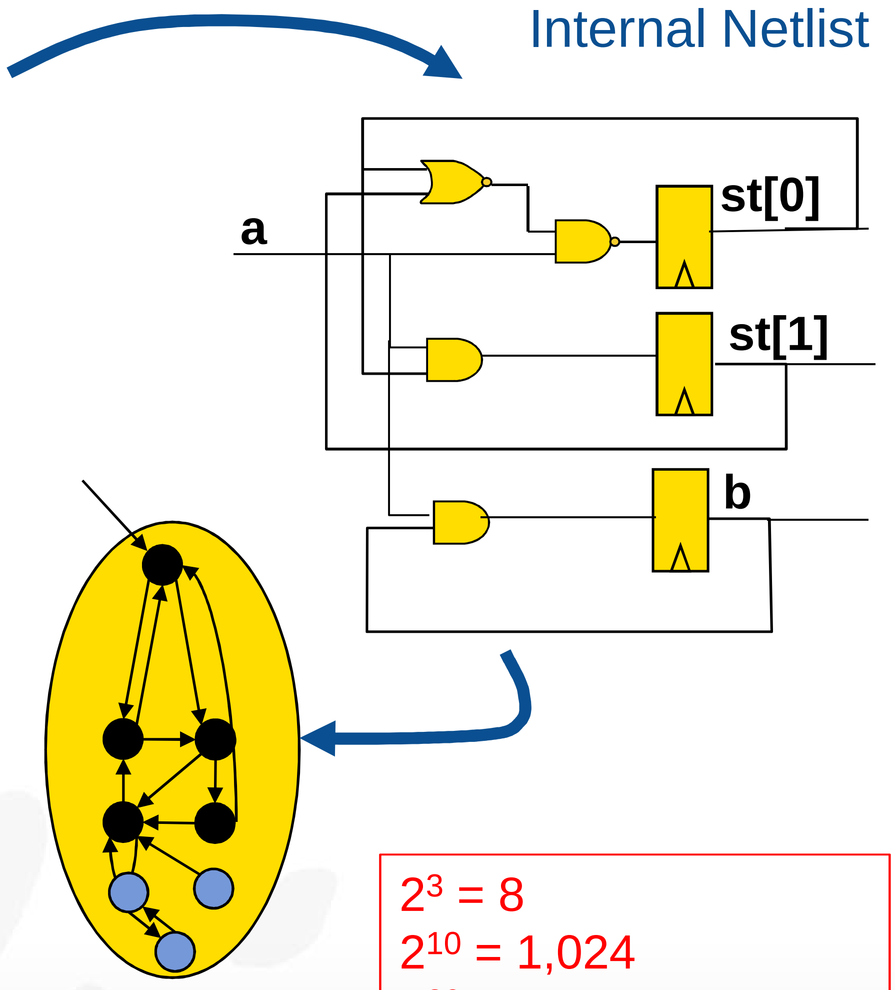
Source of Complexity

RTL

```
input a;
reg b;
reg [1:0] st;

always @(posedge clk or negedge rst)
  if (~rst) st <= 2'b00;
  else case( st )
       2'b00:  if (~a) st <= 2'b01;
       2'b01:  st <= 2'b10;
       2'b10:  if (a) st <= 2'b00;
       endcase

always @(posedge clk or negedge rst)
  if (~rst) b <= 1'b0;
  else if (~a | b) b <= 1'b0;
  else b <= 1'b1;
```



Internal Netlist

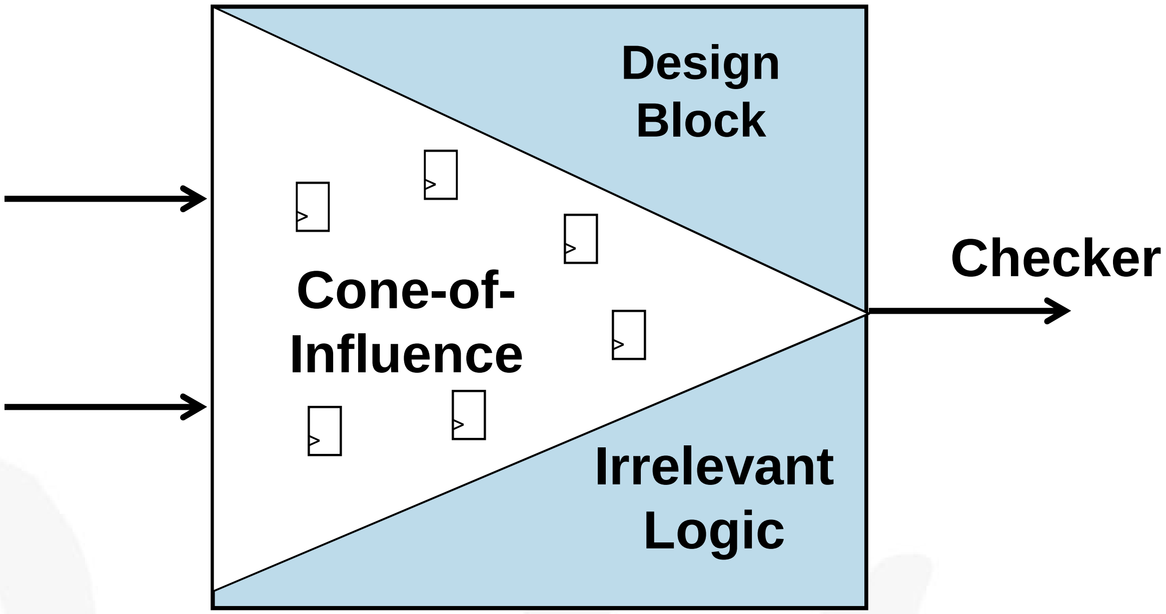
Internal STG

Checker: (st == 2'b01) => ~b

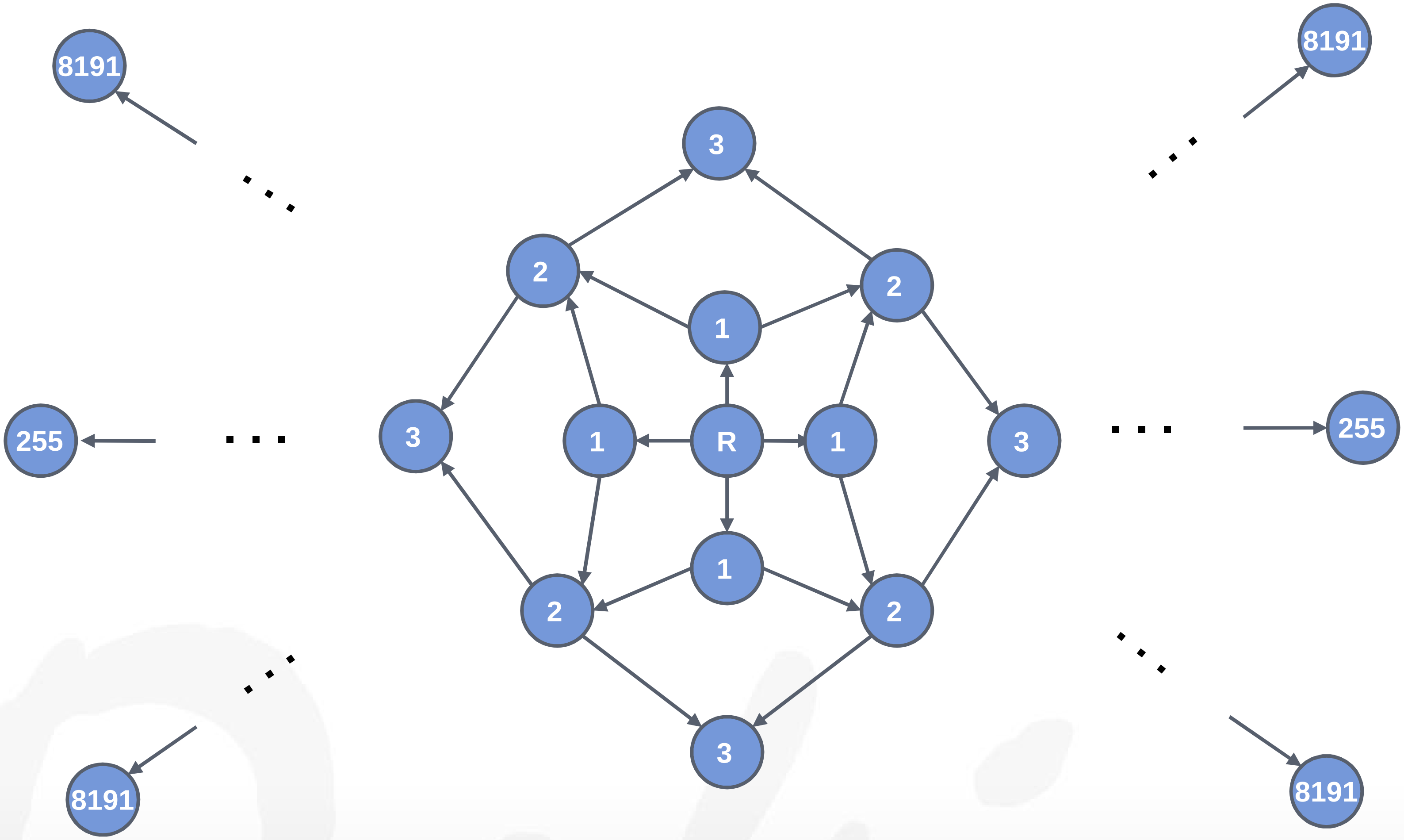
$2^3 = 8$
$2^{10} = 1{,}024$
$2^{20} = 1{,}048{,}576$
$2^{30} = 1{,}073{,}741{,}824$

Complexity – function of Cone-of-Influence

- One coarse measure of Complexity
  - number of flops/memory bits in the Cone-of-Influence of the Checker



State space complexity

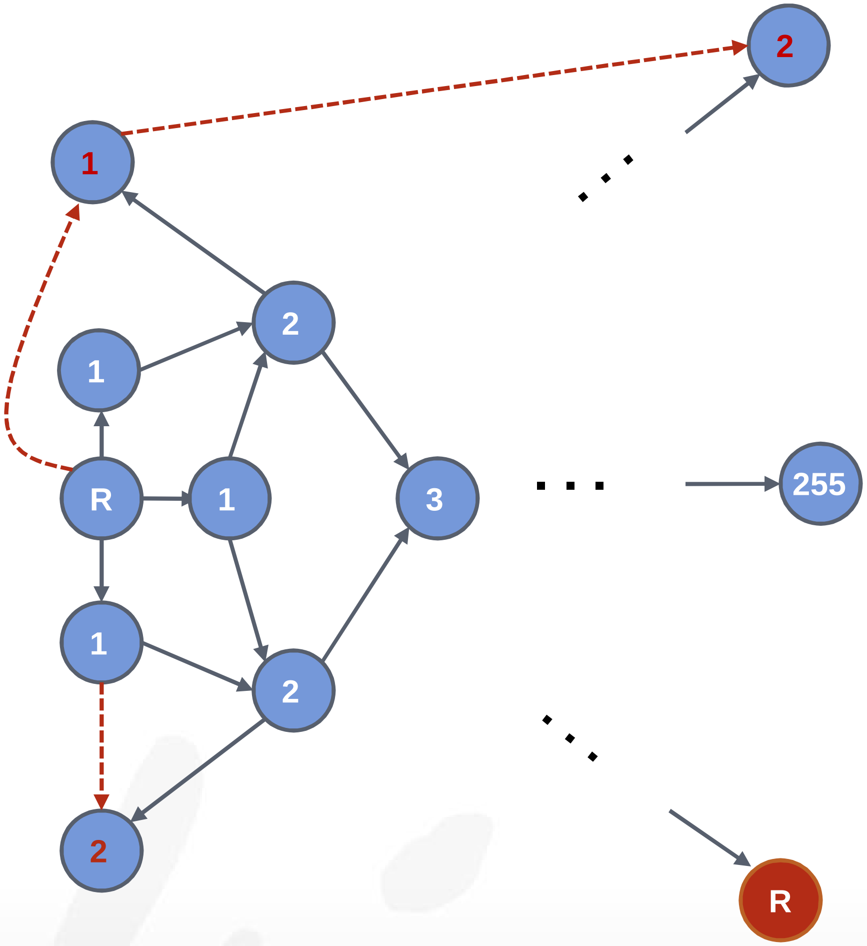


Abstractions (to manage complexity)

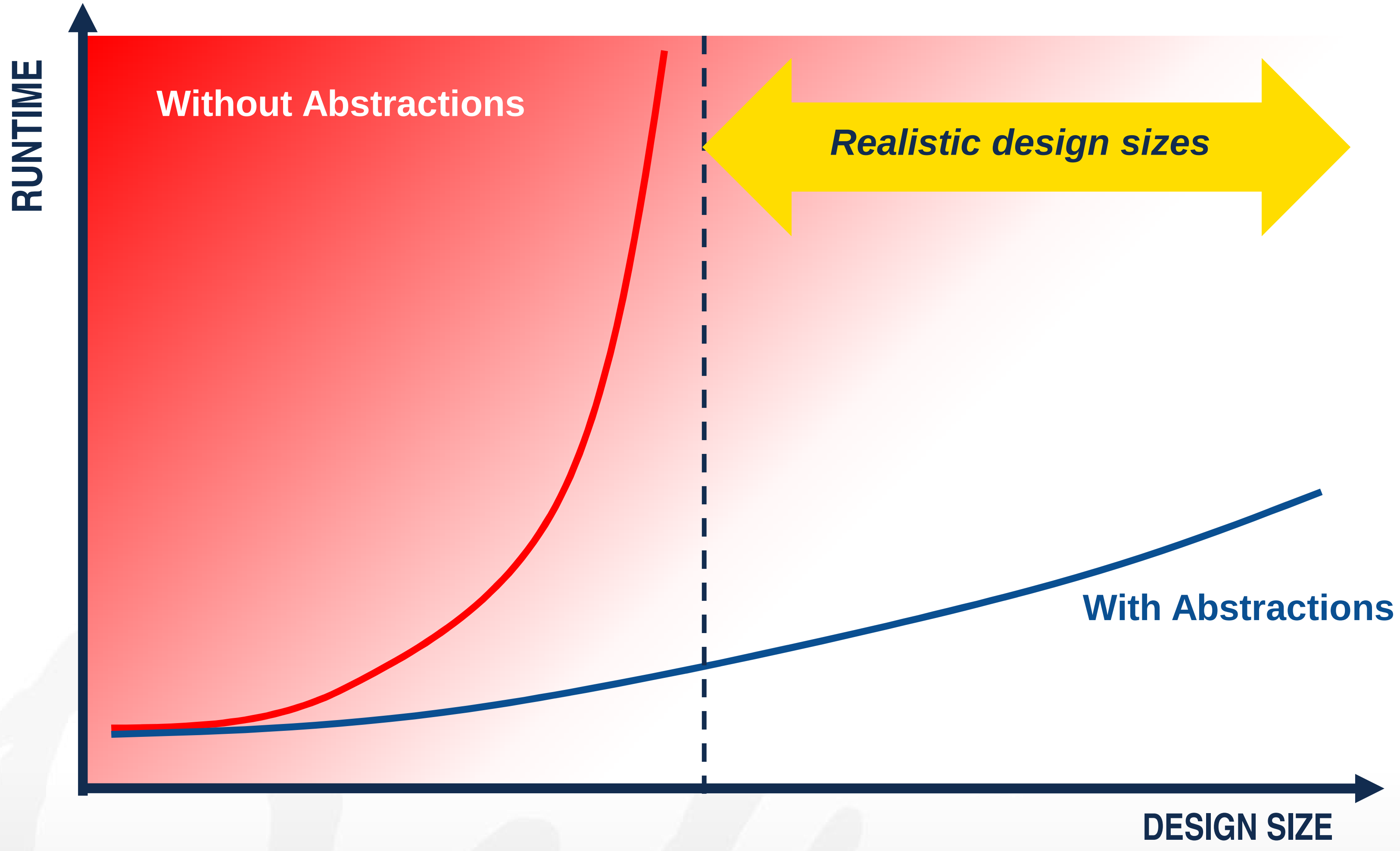
- An “Abstraction” of a design is a design that has a superset of the design behavior
- Useful to overcome complexity barriers
  - Smaller Cone-of-Influence
  - Shallower search space
  - Ability to skip long initialization sequences
- Cannot give a false positive
- Can give a false negative (Fail), but…
  - You get a trace to determine the reason for the negative

Complexity (and Abstractions)

- Effect of abstractions:
  - Reduces state space
  - Adds state transitions
  - Adds Reset states



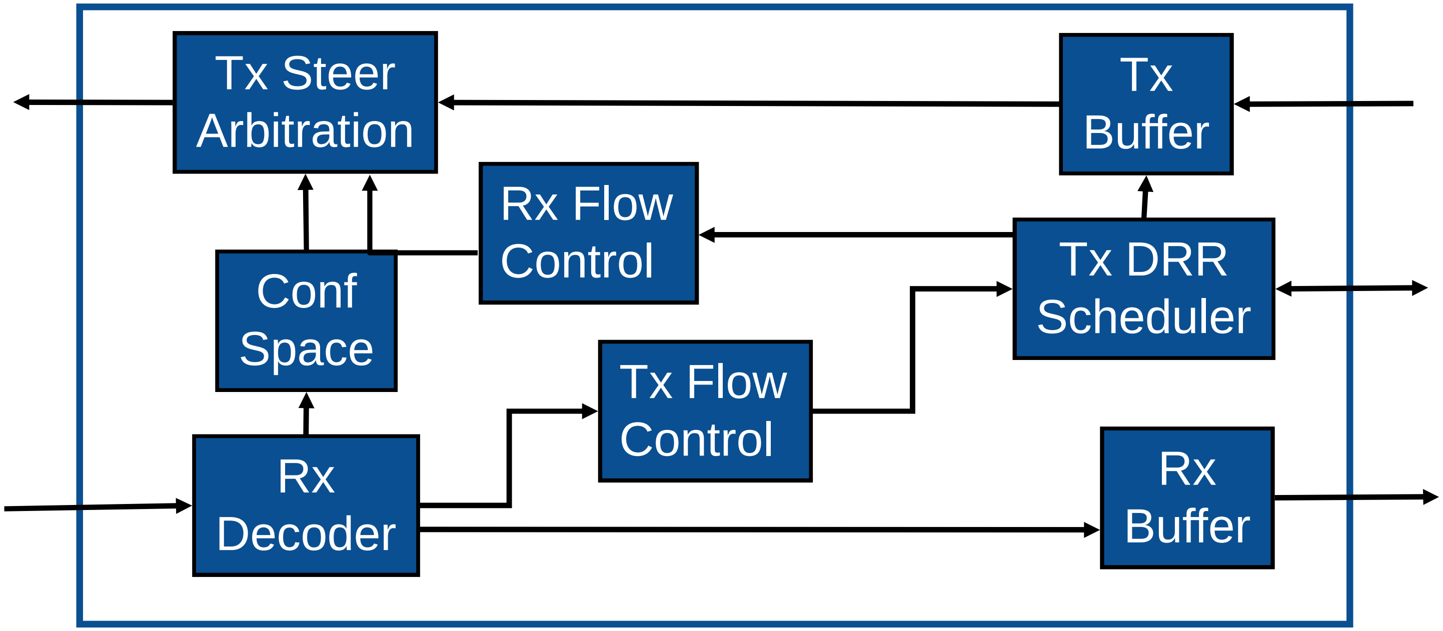
Overcoming complexity with Abstractions



Examples of Abstractions

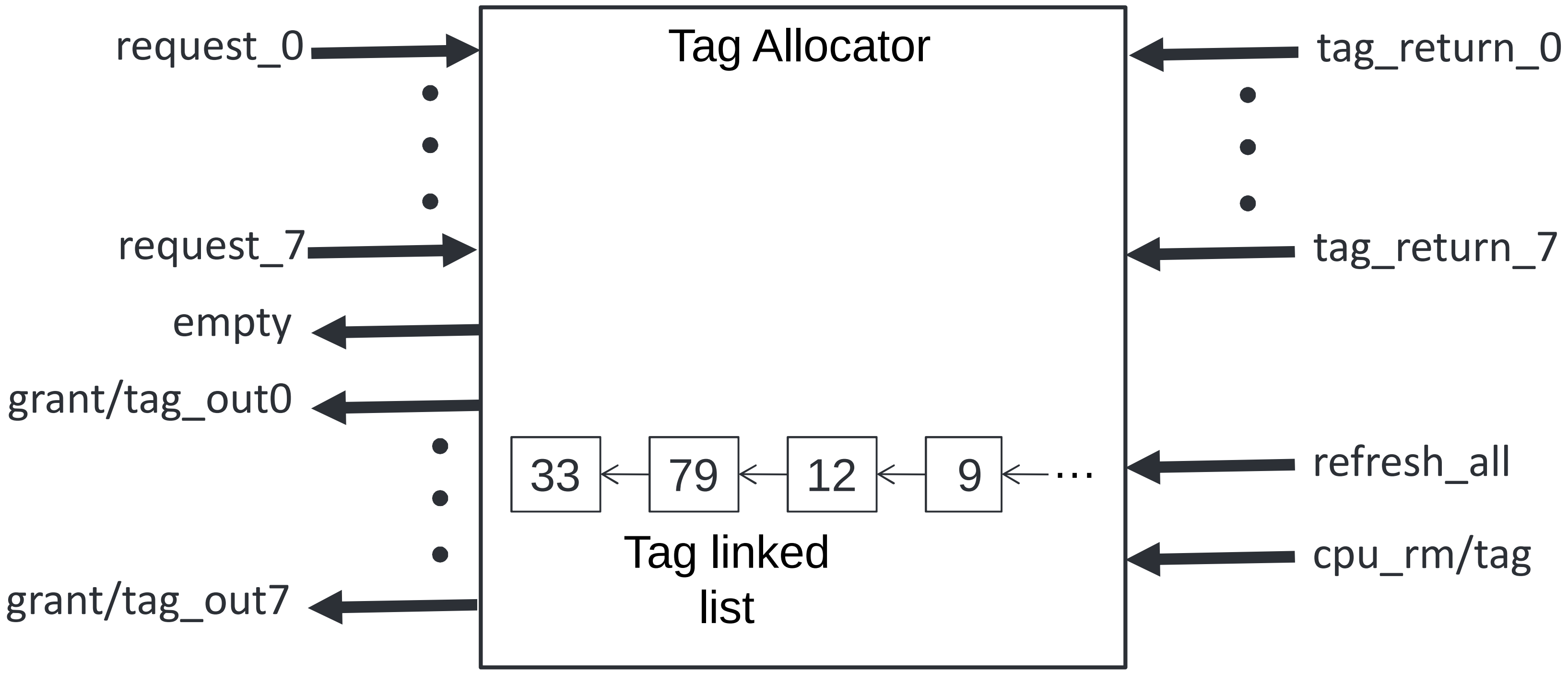
- Allow DUT to reset to a deep state
  - BANKS_IDLE state for a memory controllers (skips thousands of clocks of initialization)
- Replace memory by a memory model tracking a specific Byte of a specific Beat of a specific Transaction
  - 13th transaction, 243rd beat, 1st Byte
- Replace a Tag Generator by an abstract model
  - Reduce sequential depth by tracking specific value
  - Example in the next few slides…

Example: PCIe Transaction Layer



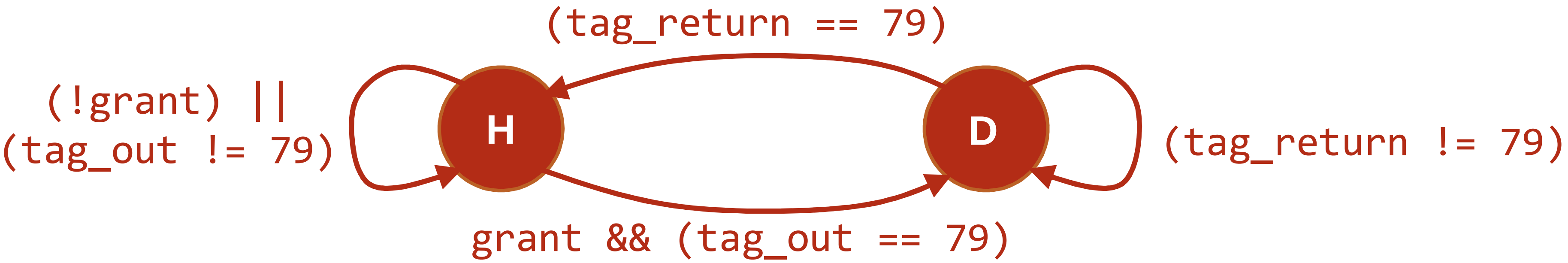
- 128-bit datapath
- 8 VCs
- History, policy-dependent DRR scheduler
- Conf responses arbitrated with TLPs and FC DLLPs
- Tx, Rx Buffers can store multiple TLPs (upto 32kb each)

Abstraction for Tag Allocator



- Pick an arbitrary, but fixed tag: e.g. tag #79
- Replace Tag Allocator by a two-state Abstraction:
  - HAS_79 (H): models that Tag_Allocator has tag #79
  - DOESNOT_HAVE_79 (D): Tag Allocator does not have #79

Abstraction for Tag Allocator



A. Use Tag Allocator Abstraction (DUT = Design with abstracted TA)

- Add constraints
  1. (state == H) |-> (!empty);
  2. (state == D) |-> (tag_out != 79);
- Add assertions:
  1. (state == H) |-> (tag_return != 79);
  2. (state == D) |-> Tag #79 is eventually returned

B. Prove Tag Allocator Abstraction (DUT = Tag Allocator RTL)

- Reverse constraints and assertions (e.g. prove no tags leak)
- Can be a sequentially long, but on a tiny DUT

Abstraction Model

- Other example of Abstraction Models:
  - Localization
  - Datapath
  - Memory
  - Sequence
  - Counter
  - Floating pulse
- Without Abstraction Models:
  - On most interesting designs, formal tools do not search far enough

Verification closure with formal and simulation

OSKI TECHNOLOGY, INC.

Unique Methodology. Highest Coverage. Fastest Time to Market.

Coverage on RTL designs

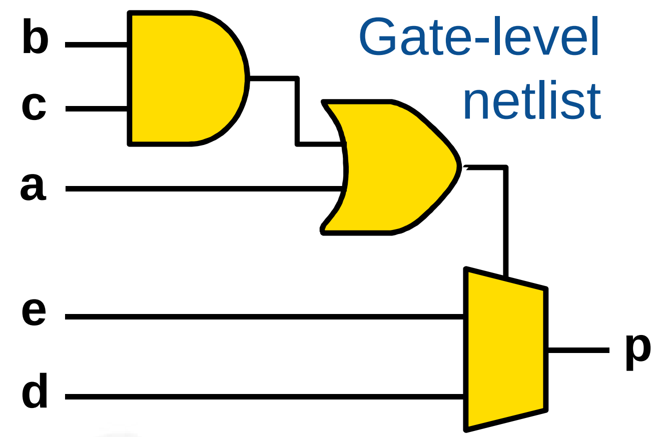
RTL (Verilog)

```
reg p;
always @(*) begin
  if (a || (b && c))
    p = d;
  else
    p = e;
end
```

Equivalent RTL

```
reg w, p;
always @(*) begin
  w = a || (b && c);
end
always @ (*) begin
  p = (w && d) || ((!w) && e);
end
```

Synthesis



Gate-level netlist

Input Coverage: line/expression coverage

```
reg p;
always @(*) begin
  if (a || (b && c))
    p = d;
  else
    p = e;
end
```

Line coverage

a	b	c	p	
0	0	0	e	**target #1**
0	0	1	e	
0	1	0	e	
0	1	1	d	**target #2**
1	0	0	d	
1	0	1	d	
1	1	0	d	
1	1	1	d	

Expression coverage

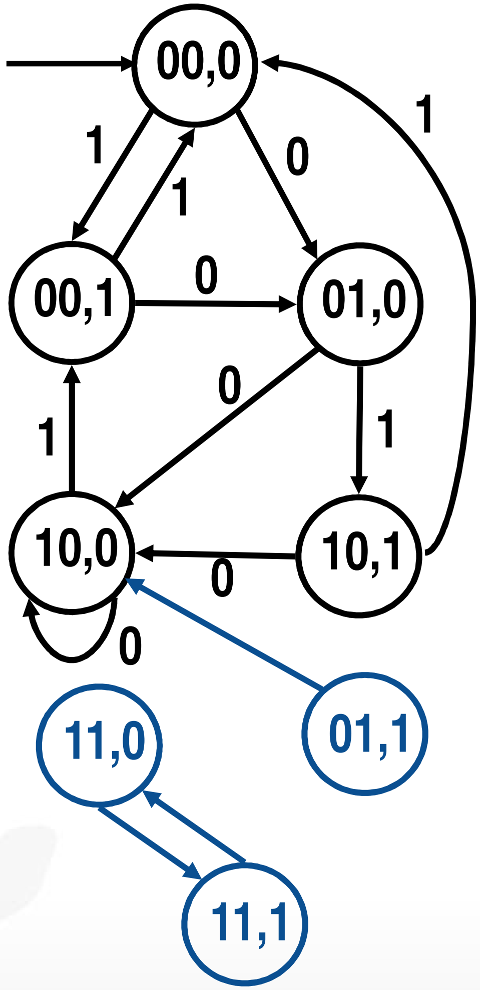
a	b	c	p	
0	0	0	e	
0	0	1	e	**#1**
0	1	0	e	**#2**
0	1	1	d	**#3**
1	0	0	d	**#4**
1	0	1	d	
1	1	0	d	
1	1	1	d	

STG coverage vs code coverage

```
input a;
reg b;
reg [1:0] st;

always @(posedge clk or negedge rst)
  if (~rst) st <= 2’b00;
  else case( st )
       2’b00:  if (~a) st <= 2’b01;
       2’b01:  st <= 2’b10;
       2’b10:  if (a) st <= 2’b00;
       endcase

always @(posedge clk or negedge rst)
  if (~rst) b <= 1’b0;
  else if (~a | b) b <= 1’b0;
  else b <= 1’b1;
```

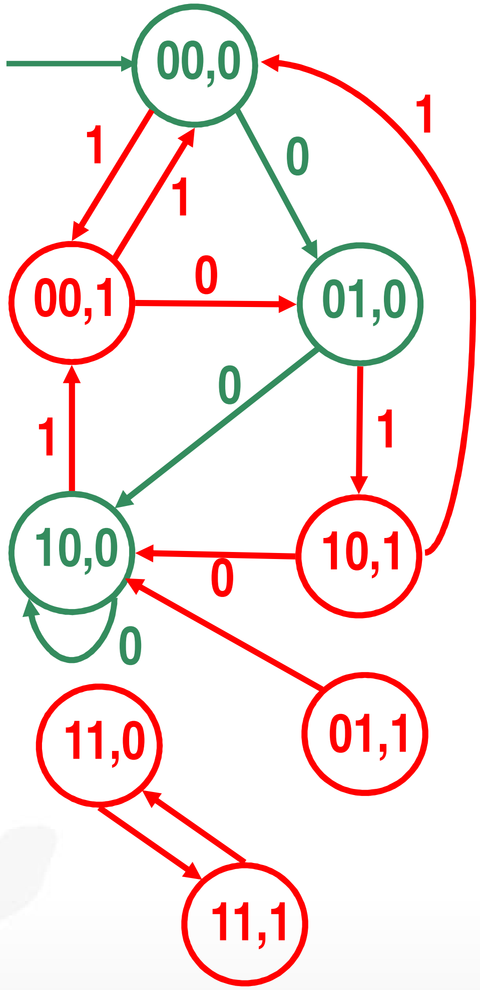


Simulation coverage (a = 0)

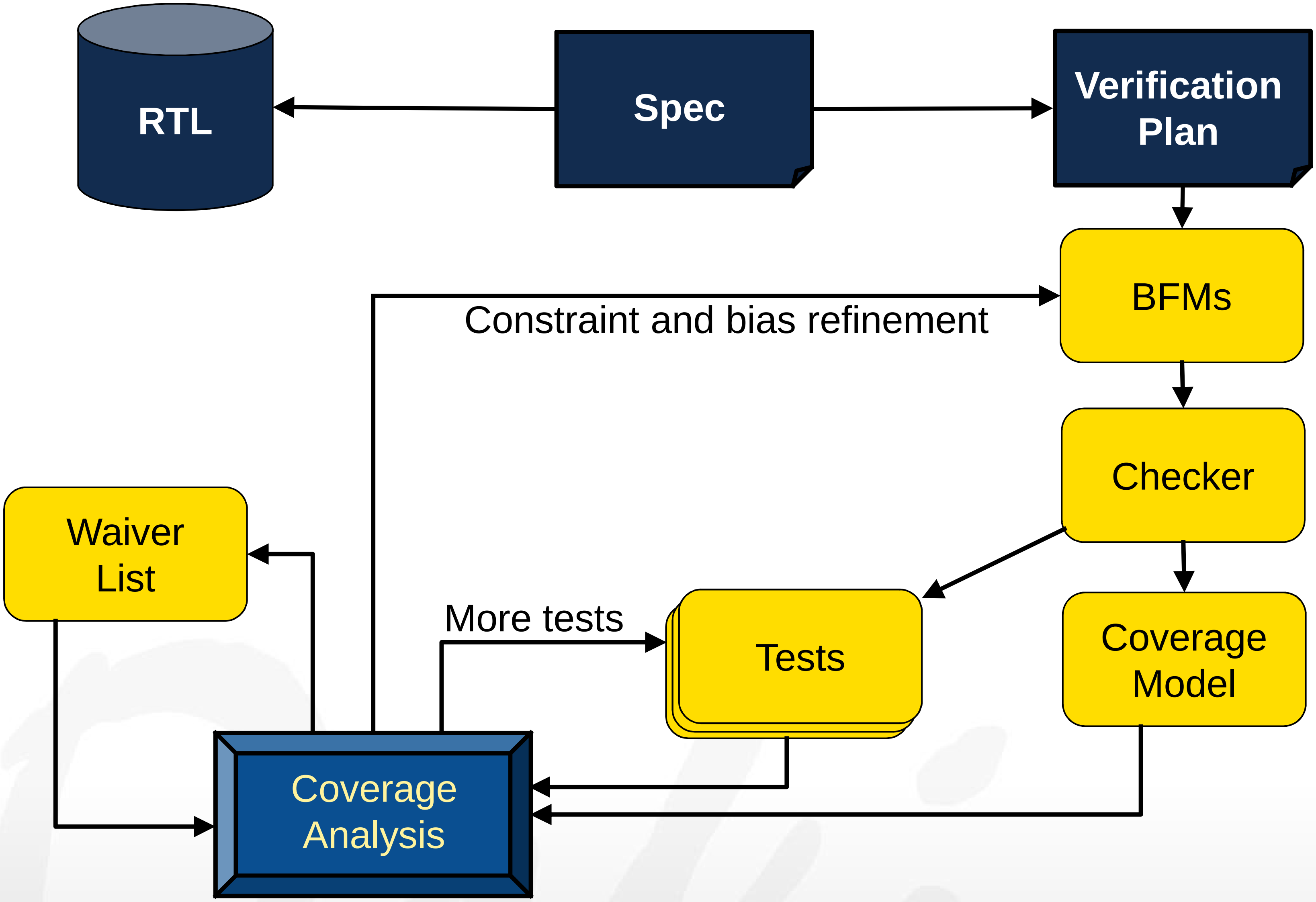
```
input a;
reg b;
reg [1:0] st;

always @(posedge clk or negedge rst)
  if (~rst) st <= 2’b00;
  else case( st )
       2’b00:  if (~a) st <= 2’b01;
       2’b01:  st <= 2’b10;
       2’b10:  if (a) st <= 2’b00;
       endcase

always @(posedge clk or negedge rst)
  if (~rst) b <= 1’b0;
  else if (~a | b) b <= 1’b0;
  else b <= 1’b1;
```



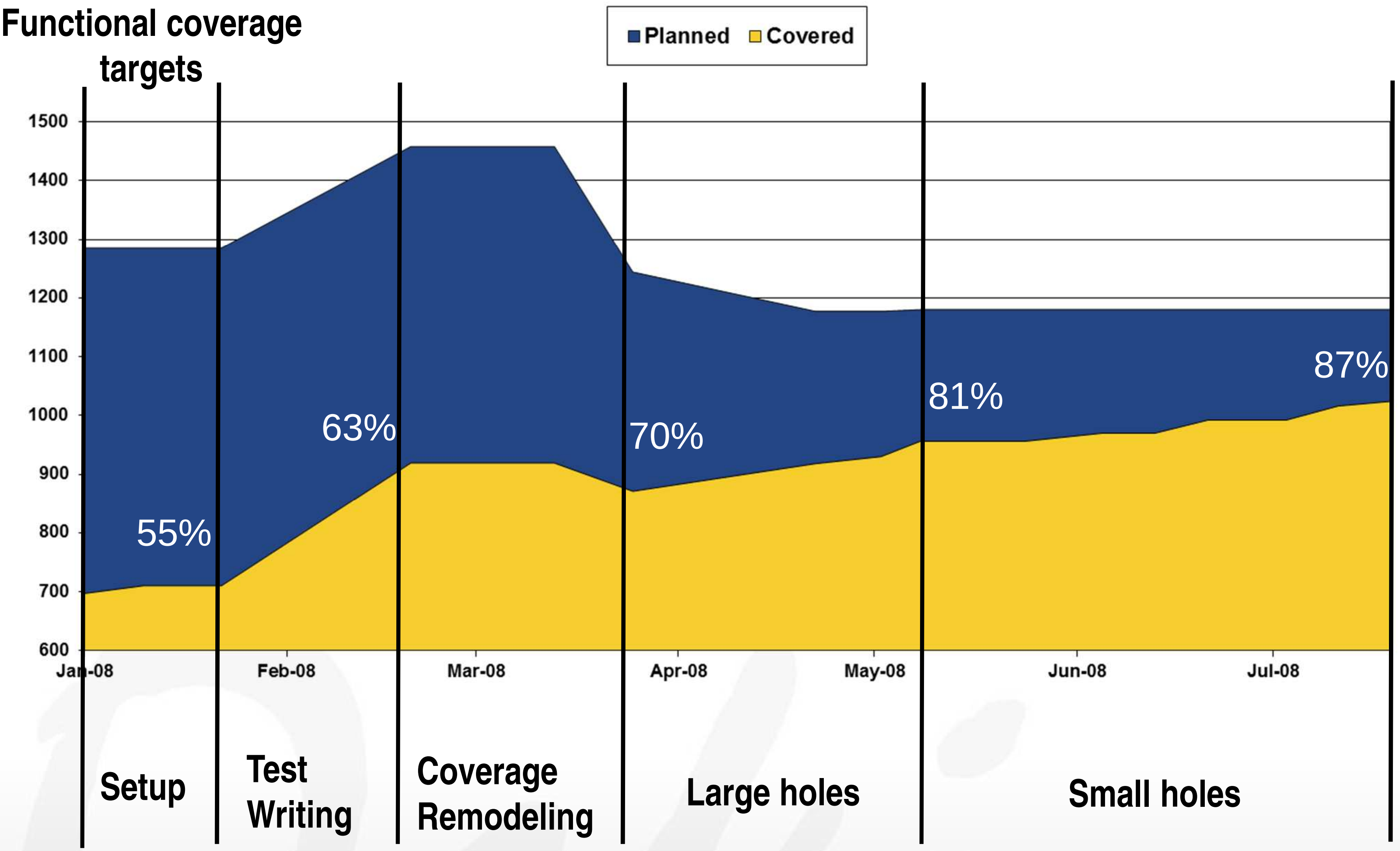
Coverage-driven simulation methodology



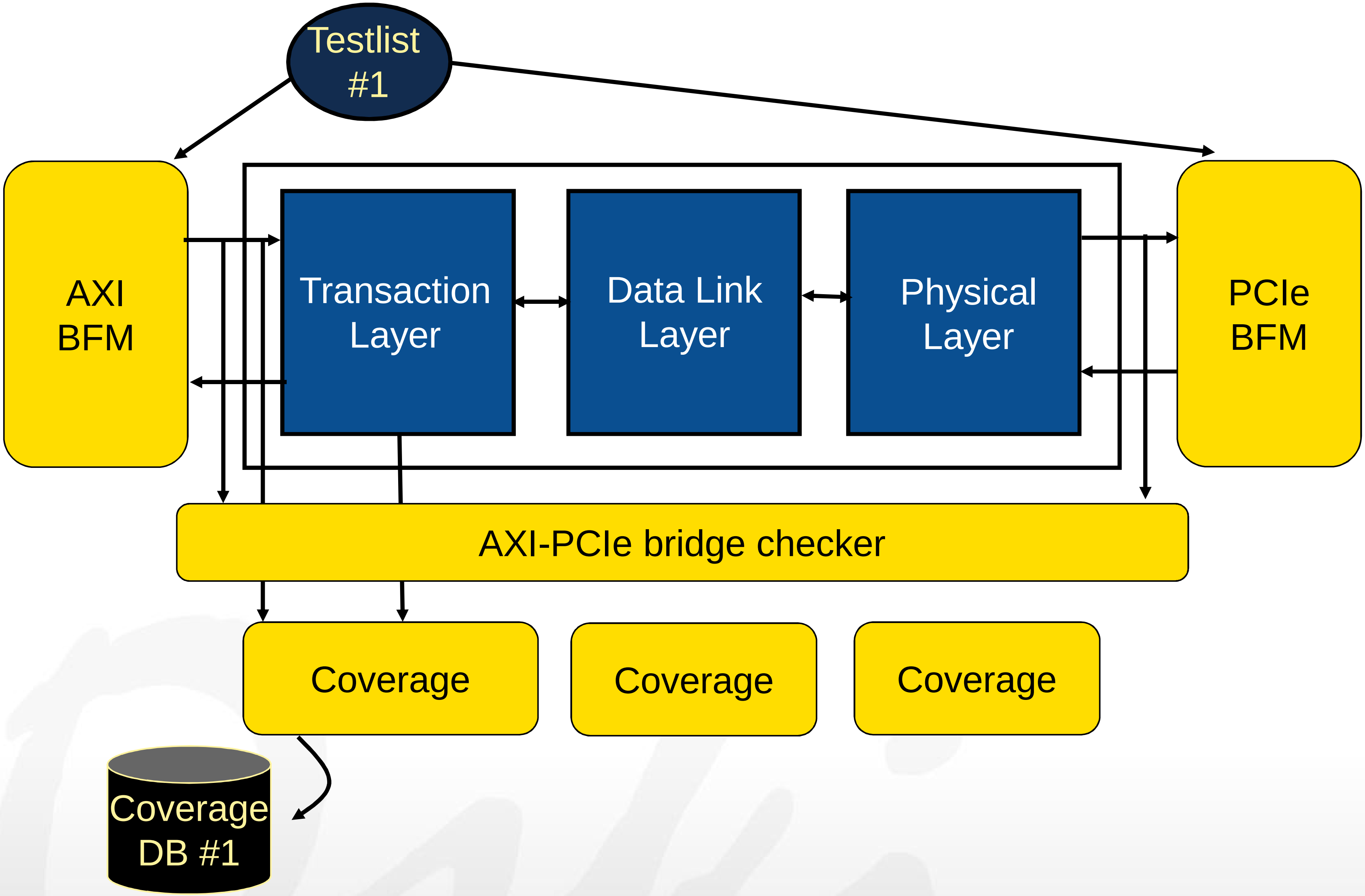
Coverage for hardware designs

- Trivial to get to 60-70% code coverage
- 100% line/expression coverage often required for tapeouts
  - Manual waivers are allowed
- NVIDIA SNUG 2011 paper
  - 270 man weeks to do waiver analysis for one design
  - 180 man weeks to write missing tests

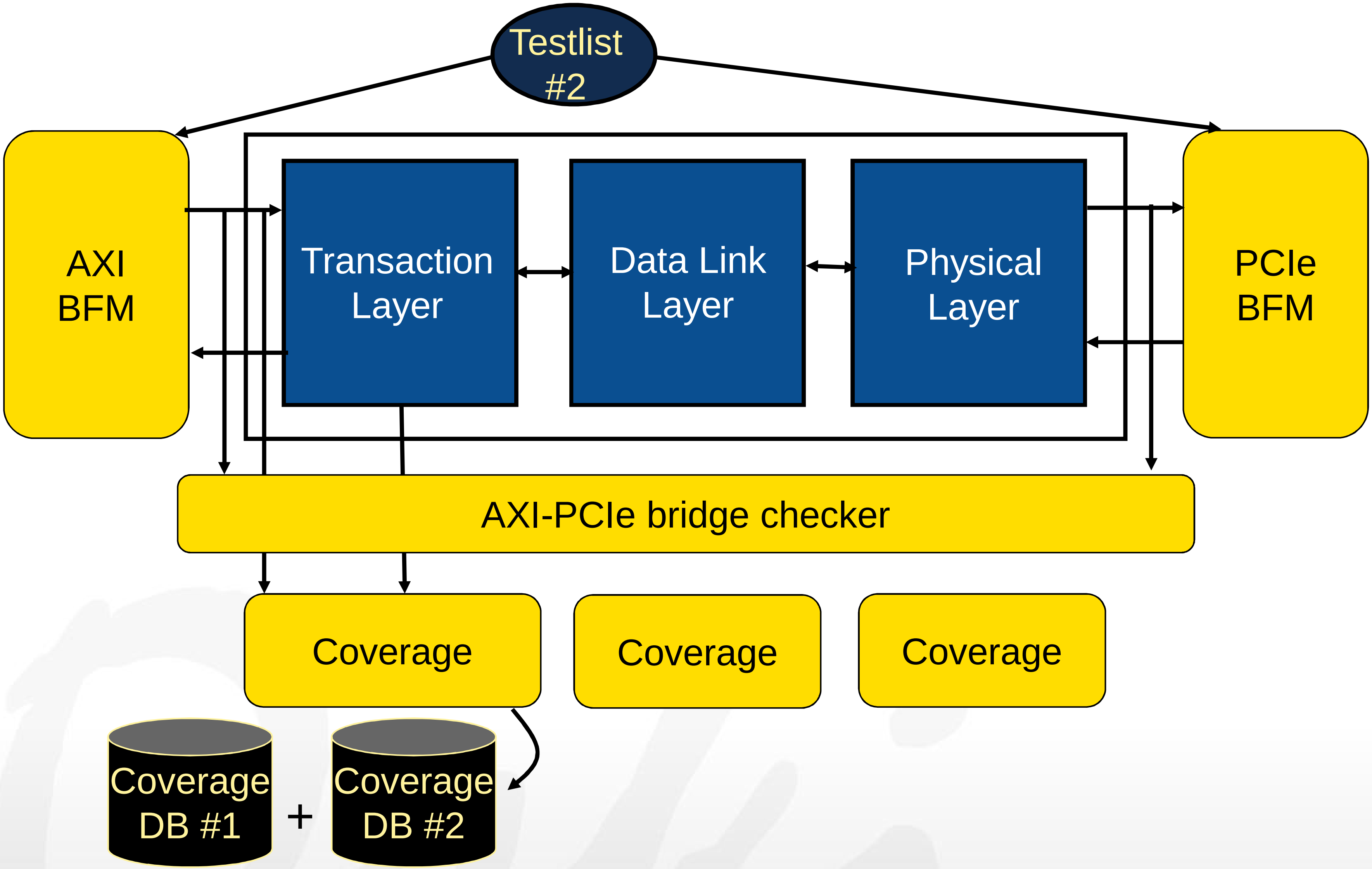
Coverage closure phases



Coverage database collection



Coverage database collection



Coverage is not the be-all and end-all

“The perfect is the enemy of good”

-Voltaire (1772)

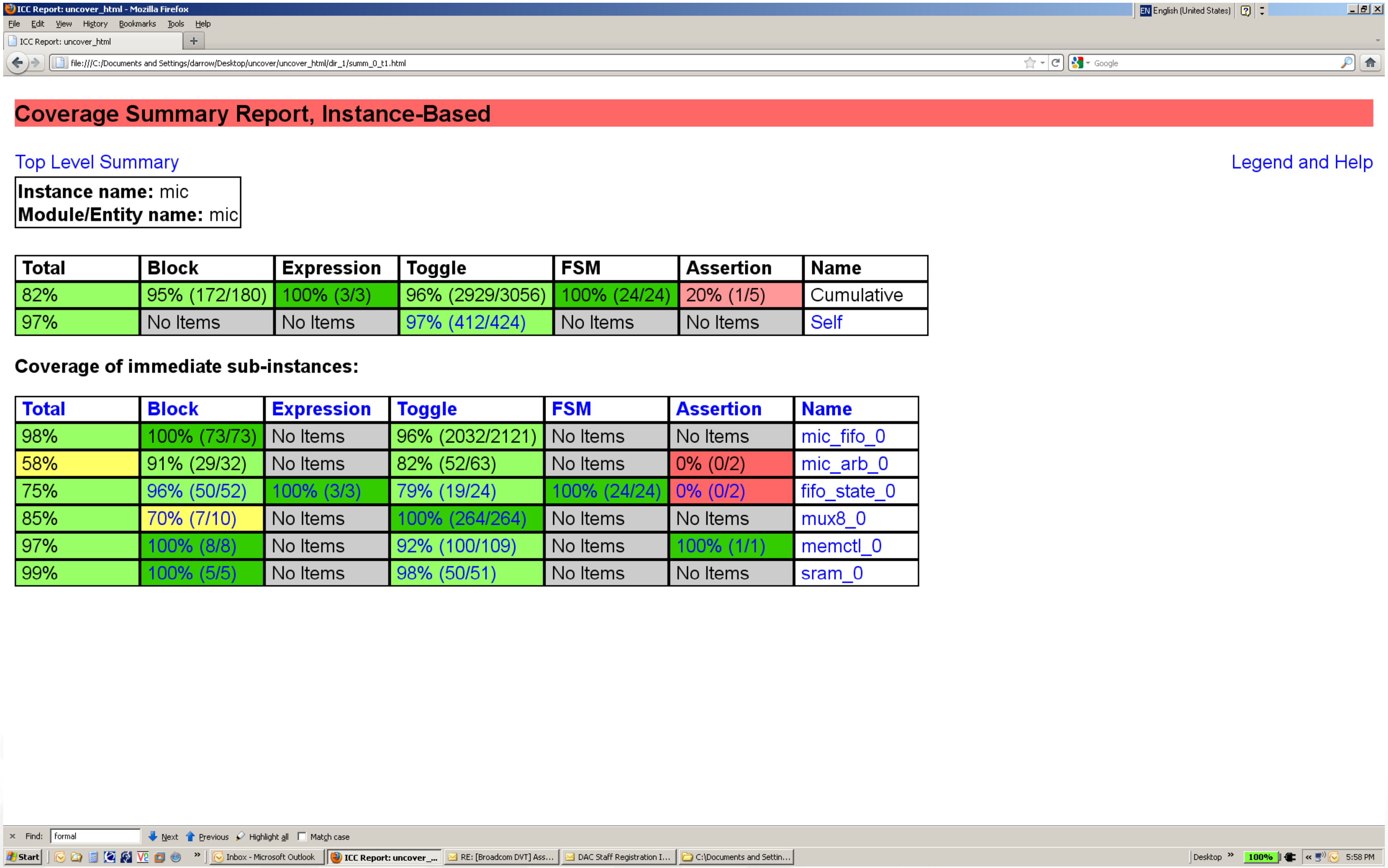
- Coverage is not perfect
  - Bugs are missed even with 100% coverage
- But…
  - Helps measure progress
  - Helps identify blind-spots

Input vs Observable coverage

- “Have I verified enough input sequences” (Input coverage)
- “Is my set of checkers complete enough” (Observable coverage)
- Same two notions apply for both simulation AND formal
  - Bounded model checking (BMC) is the most used formal technique

NOTE ***Formal does not verify all possible input sequences***

Coverage reporting


ICC Report: uncover_html - Mozilla Firefox
English (United States)
File Edit View History Bookmarks Tools Help
ICC Report: uncover_html
file:///C:/Documents and Settings/darrow/Desktop/uncover/uncover_html/dir_1/summ_0_t1.html
Google
Find: formal
Next
Previous
Highlight all
Match case
Start
Inbox - Microsoft Outlook
ICC Report: uncover_...
RE: [Broadcom DVT] Ass...
DAC Staff Registration I...
C:\Documents and Settin...
Desktop
100%
5:58 PM

Coverage Summary Report, Instance-Based

Top Level Summary

Legend and Help

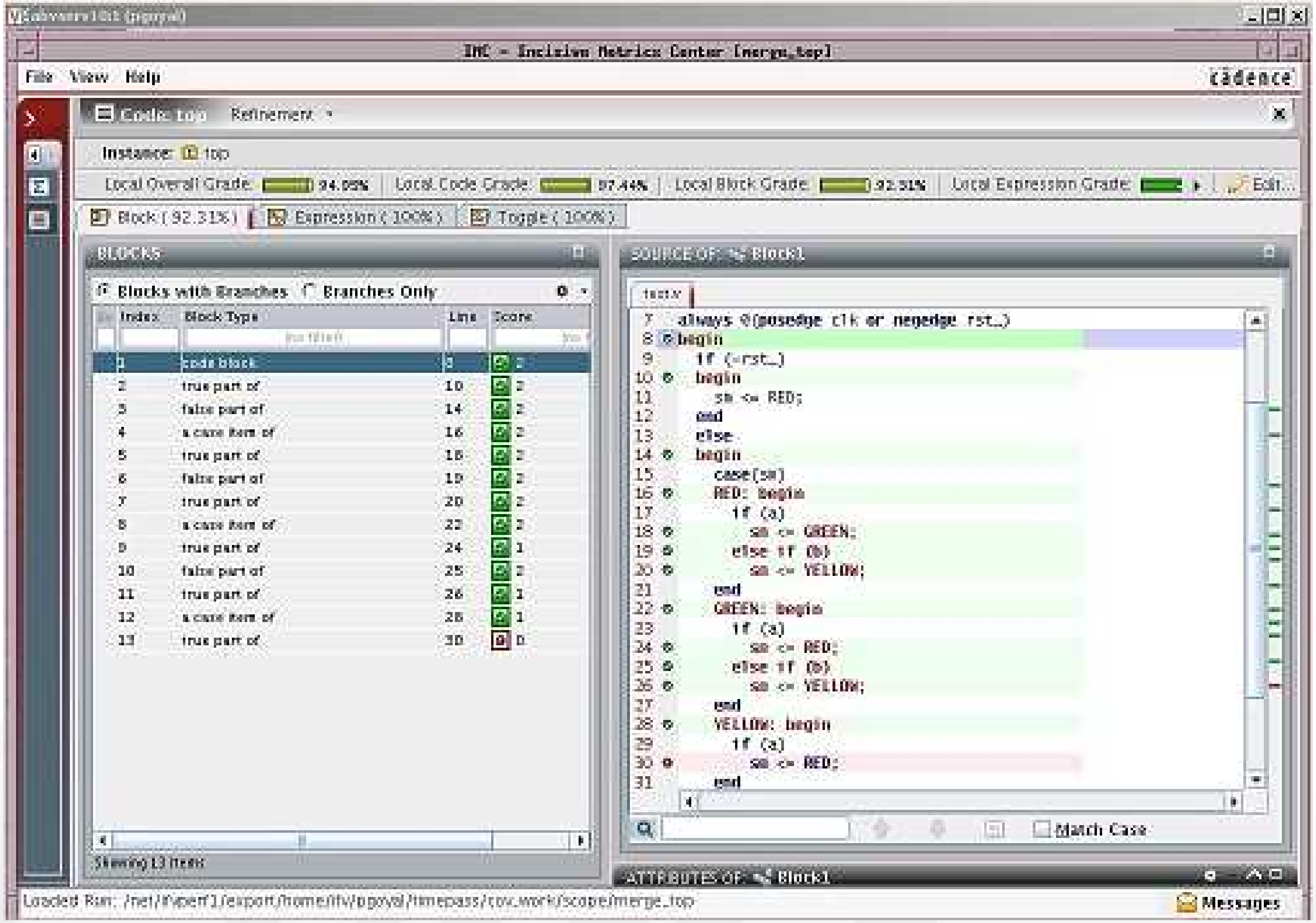
Instance name: mic
Module/Entity name: mic

Total	Block	Expression	Toggle	FSM	Assertion	Name
82%	95% (172/180)	100% (3/3)	96% (2929/3056)	100% (24/24)	20% (1/5)	Cumulative
97%	No Items	No Items	97% (412/424)	No Items	No Items	Self

Coverage of immediate sub-instances:

Total	Block	Expression	Toggle	FSM	Assertion	Name
98%	100% (73/73)	No Items	96% (2032/2121)	No Items	No Items	mic_fifo_0
58%	91% (29/32)	No Items	82% (52/63)	No Items	0% (0/2)	mic_arb_0
75%	96% (50/52)	100% (3/3)	79% (19/24)	100% (24/24)	0% (0/2)	fifo_state_0
85%	70% (7/10)	No Items	100% (264/264)	No Items	No Items	mux8_0
97%	100% (8/8)	No Items	92% (100/109)	No Items	100% (1/1)	memctl_0
99%	100% (5/5)	No Items	98% (50/51)	No Items	No Items	sram_0

Coverage reporting



Instance: top

Local Overall Grade 94.05% | Local Code Grade 97.44% | Local Block Grade 92.31% | Local Expression Grade

Block (92.31%) | Expression (100%) | Toggle (100%)

● Blocks with Branches ○ Branches Only

Index	Block Type	Line	Score
1	code block	8	🟩 2
2	true part of	10	🟩 2
3	false part of	14	🟩 2
4	a case item of	16	🟩 2
5	true part of	18	🟩 2
6	false part of	19	🟩 2
7	true part of	20	🟩 2
8	a case item of	22	🟩 2
9	true part of	24	🟩 1
10	false part of	25	🟩 2
11	true part of	26	🟩 1
12	a case item of	28	🟩 1
13	true part of	30	🟥 0

Showing 13 items

test.v

```
always @(posedge clk or negedge rst_)
begin
  if (~rst_)
  begin
    sm <= RED;
  end
  else
  begin
    case(sm)
    RED: begin
      if (a)
        sm <= GREEN;
      else if (b)
        sm <= YELLOW;
    end
    GREEN: begin
      if (a)
        sm <= RED;
      else if (b)
        sm <= YELLOW;
    end
    YELLOW: begin
      if (a)
        sm <= RED;
    end
```

Loaded Run: /net/if/xperf1/export/home/ifv/pgoyal/timepass/cov_work/scope/merge_top

Is my formal complete?

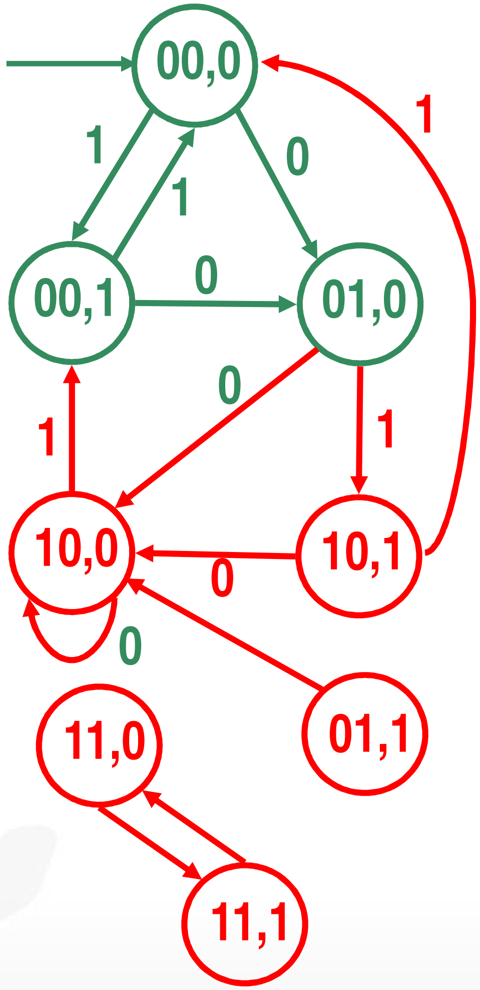
- Are my Checkers complete?
- Are my Constraints complete?
- Is my Complexity strategy complete?

Formal coverage (depth = 1)

```
input a;
reg b;
reg [1:0] st;

always @(posedge clk or negedge rst)
  if (~rst) st <= 2'b00;
  else case( st )
       2'b00:  if (~a) st <= 2'b01;
       2'b01:  st <= 2'b10;
       2'b10:  if (a) st <= 2'b00;
       endcase

always @(posedge clk or negedge rst)
  if (~rst) b <= 1'b0;
  else if (~a | b) b <= 1'b0;
  else b <= 1'b1;
```



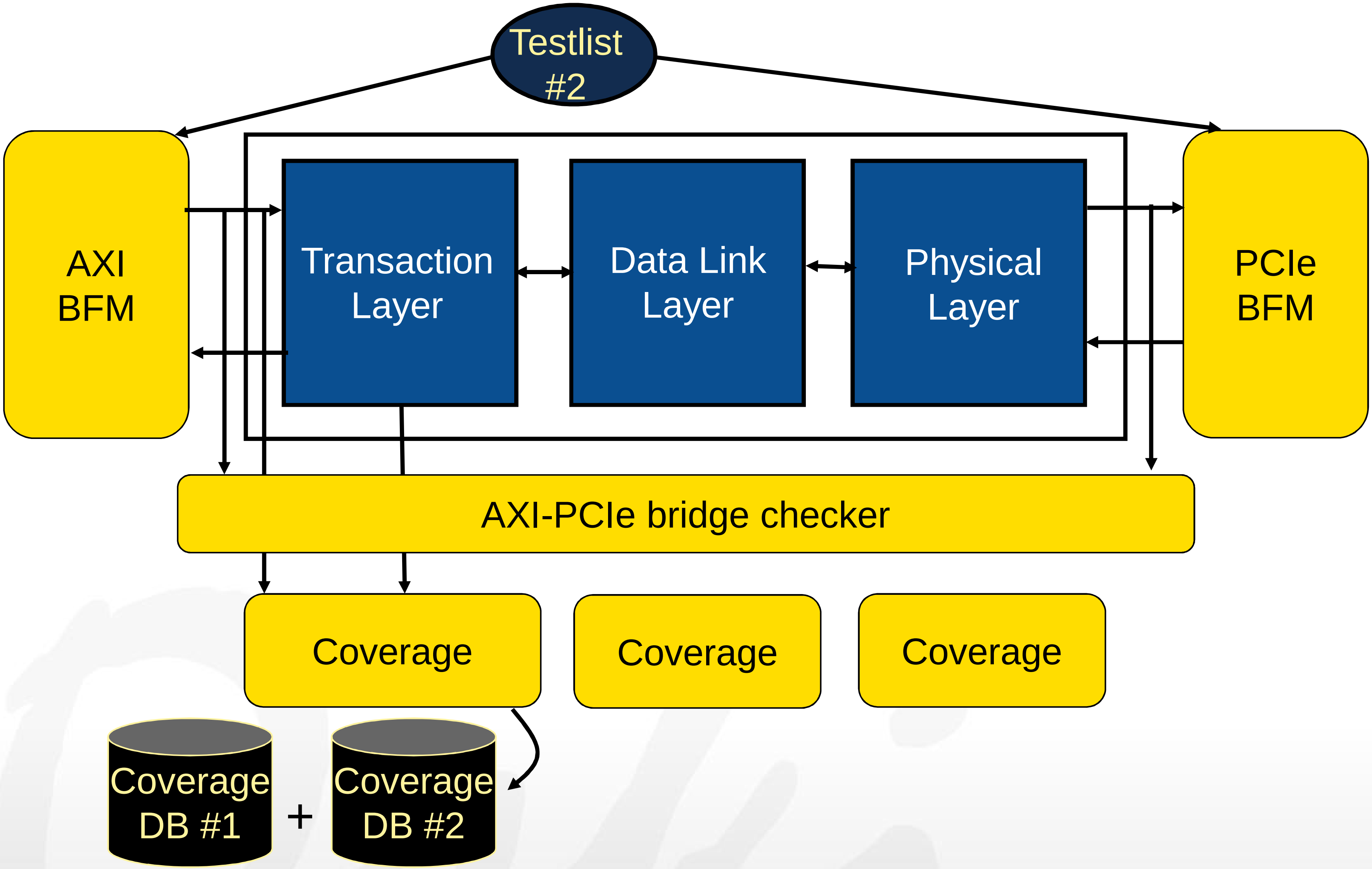
Input Coverage for formal

- Constraints: Environment may be over-constrained
  - Intentional: avoided some hard to model or verify input combinations
  - Unintentional: bugs in constraints; forgot to remove intentional over-constraints

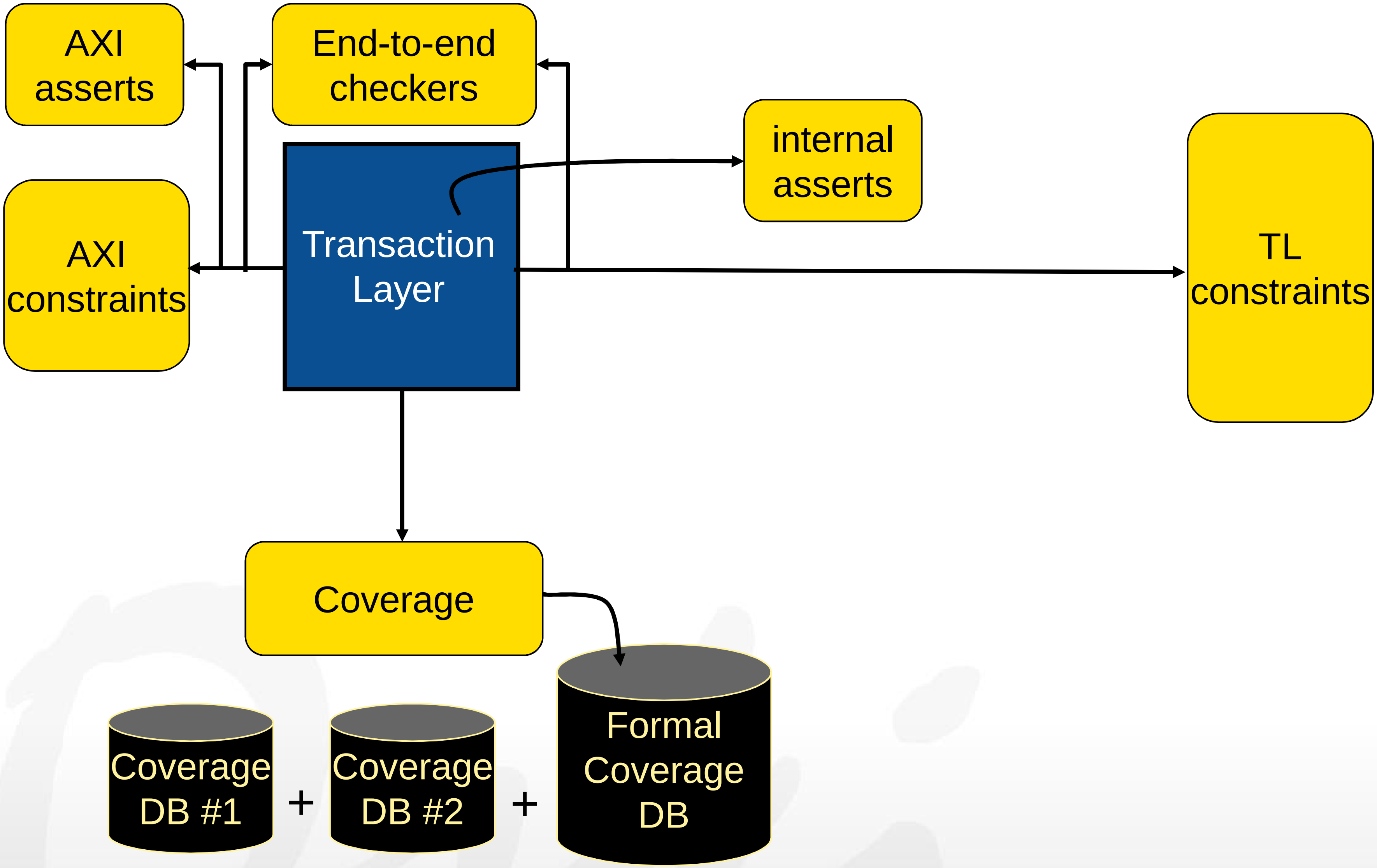
- Complexity: All checkers are verified up to proof depth N
  - Any target, not reachable in N clocks, is not covered

- Checkers: does not verify completeness of Checkers
  - No different than simulation!

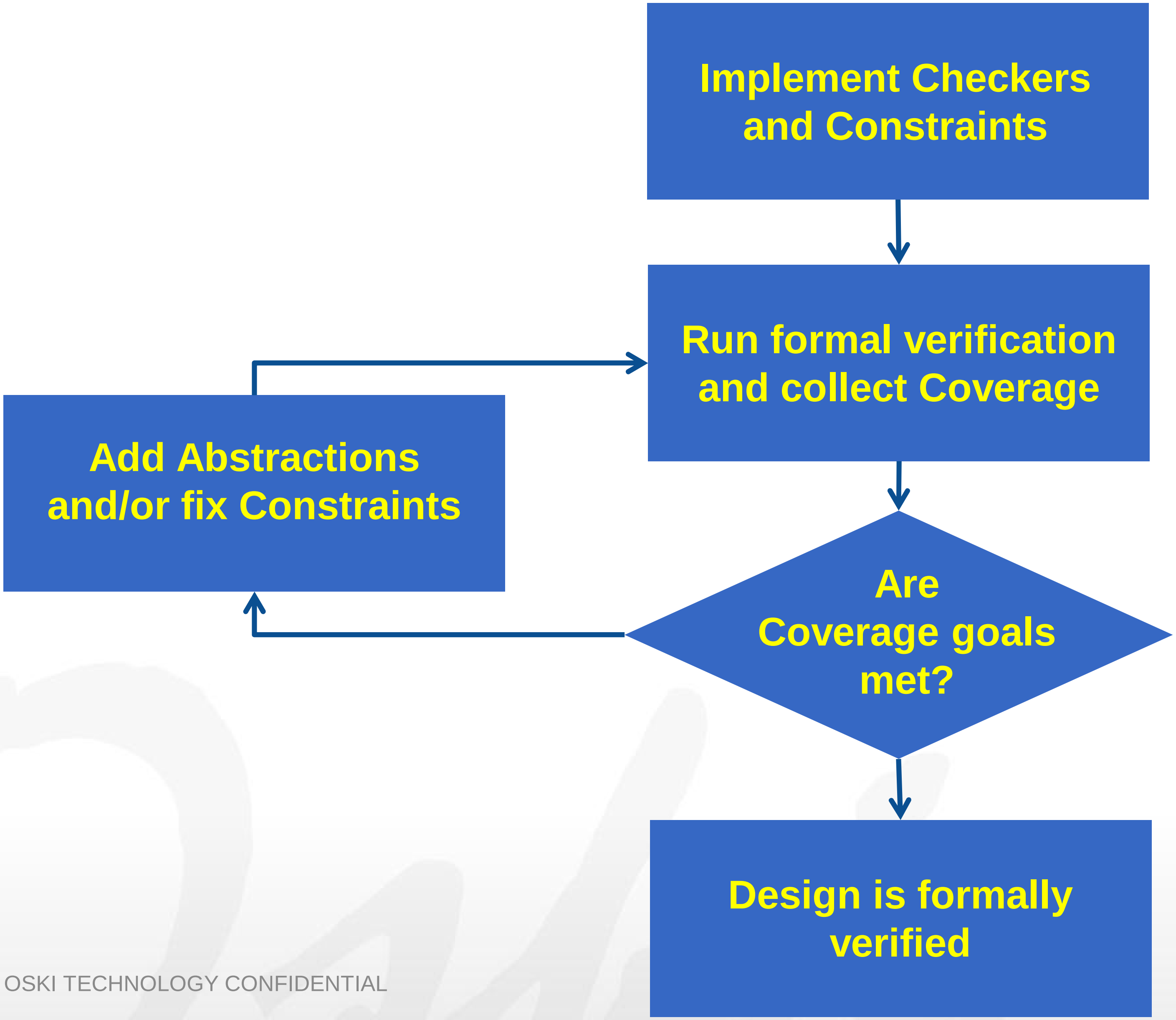
Coverage database collection



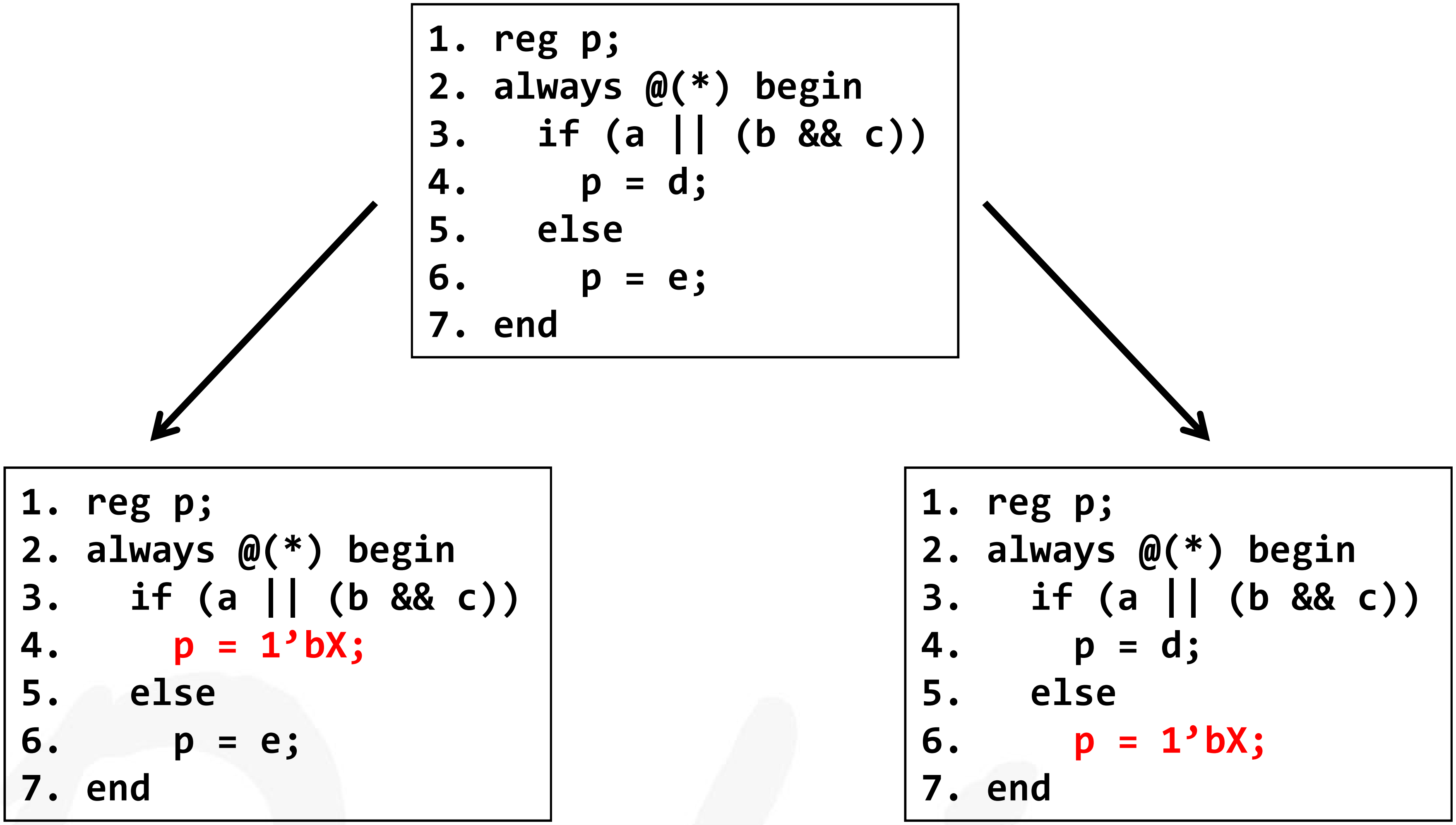
Formal coverage integrated with simulation



Formal code coverage methodology


Implement Checkers and Constraints
Run formal verification and collect Coverage
Add Abstractions and/or fix Constraints
Are Coverage goals met?
Design is formally verified

Observable Coverage (using mutations)



```
1. reg p;
2. always @(*) begin
3.   if (a || (b && c))
4.     p = d;
5.   else
6.     p = e;
7. end
```

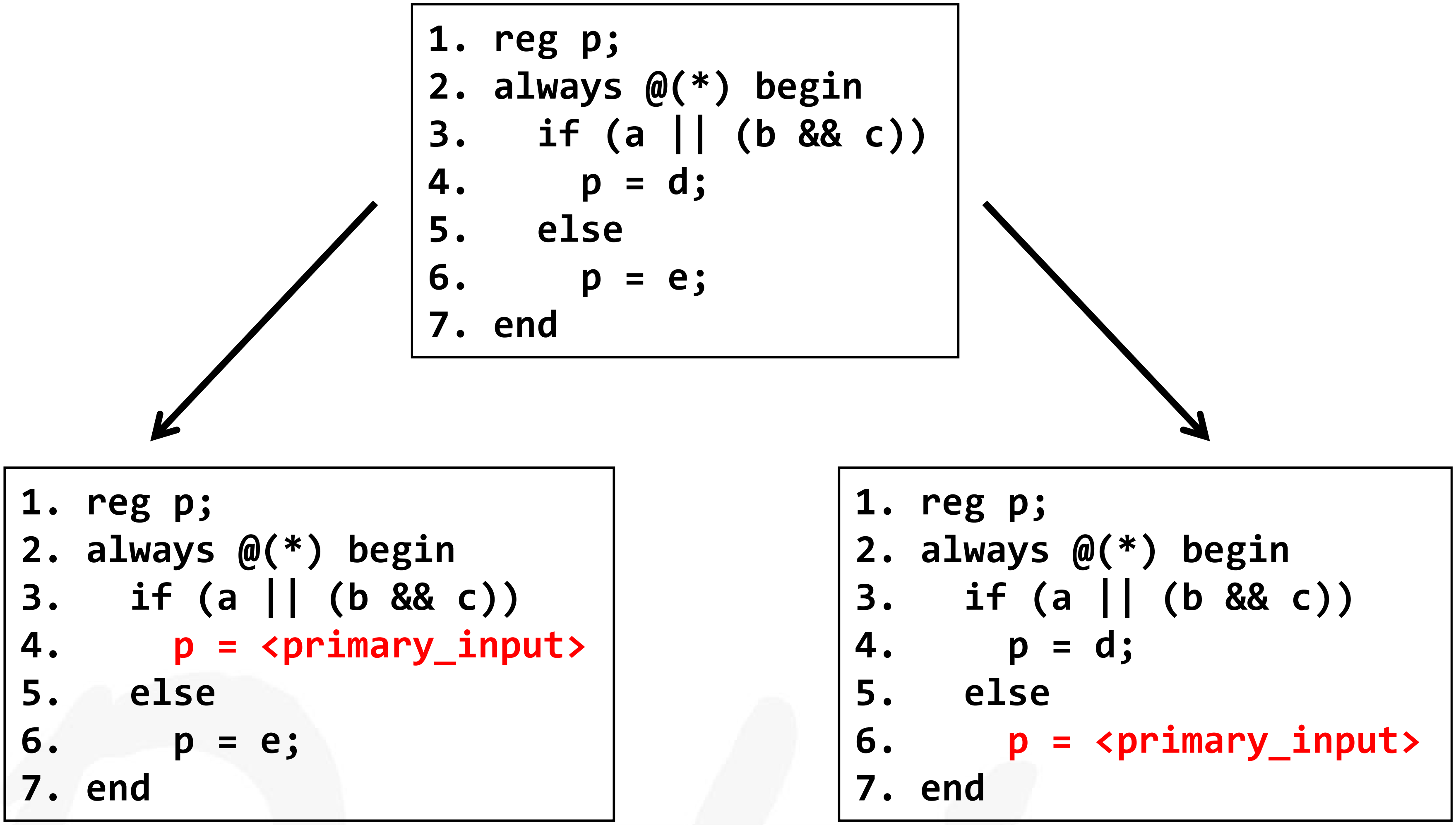
```
1. reg p;
2. always @(*) begin
3.   if (a || (b && c))
4.     p = 1’bX;
5.   else
6.     p = e;
7. end
```

Mutant for line#4

```
1. reg p;
2. always @(*) begin
3.   if (a || (b && c))
4.     p = d;
5.   else
6.     p = 1’bX;
7. end
```

Mutant for line#6

Observable Coverage for formal



```
1. reg p;
2. always @(*) begin
3.   if (a || (b && c))
4.     p = d;
5.   else
6.     p = e;
7. end
```

```
1. reg p;
2. always @(*) begin
3.   if (a || (b && c))
4.     p = <primary_input>
5.   else
6.     p = e;
7. end
```

Mutant for line#4

```
1. reg p;
2. always @(*) begin
3.   if (a || (b && c))
4.     p = d;
5.   else
6.     p = <primary_input>
7. end
```

Mutant for line#6

Conclusions

- Formal Coverage must fit with Simulation Coverage
  - Same metrics, same meaning

- Formal verification in practice:
  - BMC is the primary workhorse of practical formal verification
  - Checkers are complex Verilog, simple SVA
  - Abstraction Models are key to increasing formal coverage

Thanks

- Adnan Aziz
- Sandesh Borgaonkar
- Richard Boulton
- Choon Chng
- Harry Foster
- Vineet Gupta
- Anton Lopatinsky
- Deepak Pant
- Philippa Slayton
- Shashidhar Thakur

Contact email: vigyan@oskitech.com