

Can Mutation Analysis Help Fix Our Broken Coverage Metrics?

Brian Bailey

Email: brian_bailey@acm.org

Tel: 503 632 7448

Cell: 503 753 6040

Web: brianbailey.us



HVC'08

Outline

- **Verification basics**
- **Existing coverage metrics**
- **Some recent advances**
- **2 small case studies**
- **Futures and open issues**

Verification definition

"Confirmation by examination and provisions of objective evidence that specified requirements have been fulfilled."

IEEE Definition of verification.

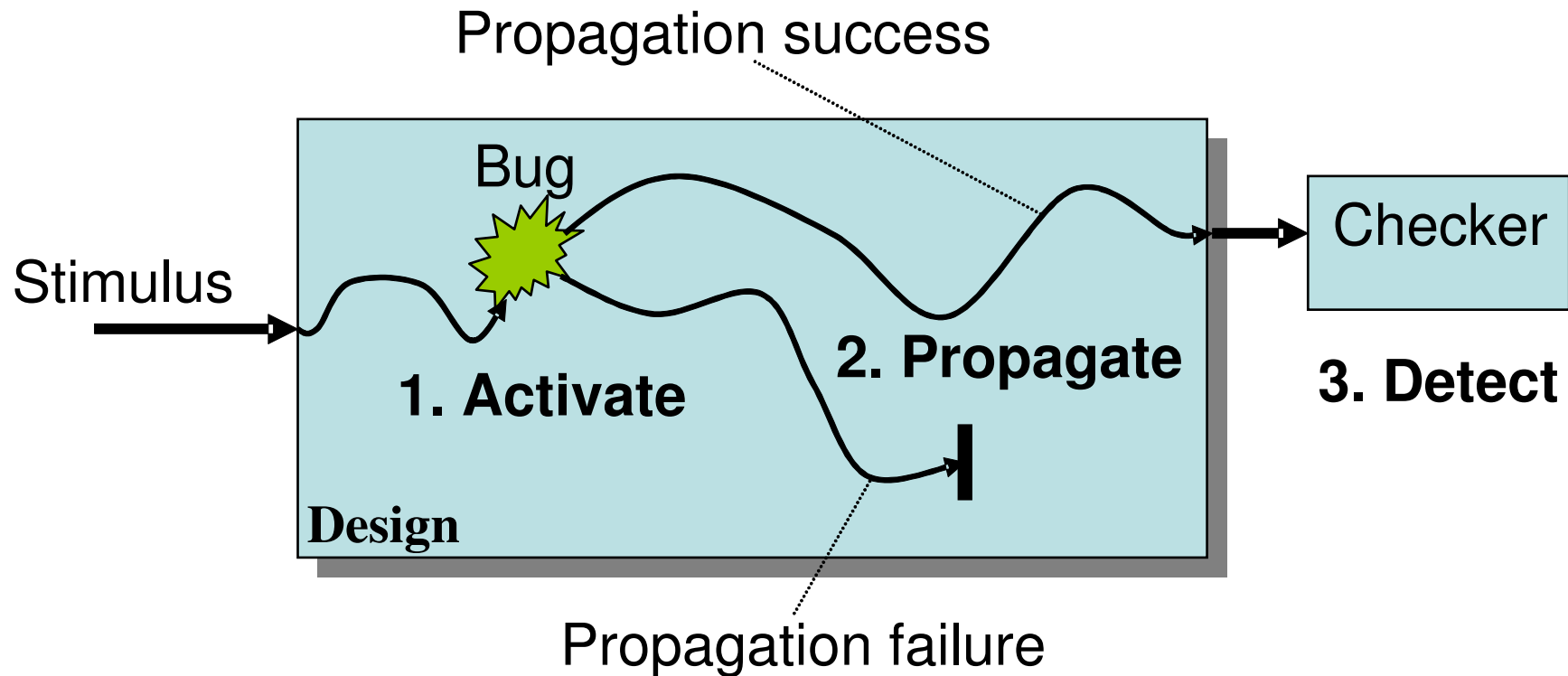
 **Verification is all about answering the question:**

Have we implemented something correctly ?

Verification definition

- 4 Key phrases
 - *Confirmation by examination*
 - If you don't look, then there is no hope of finding incorrect behavior
 - *provisions of objective evidence*
 - Requires a second independent model of functionality
 - *specified requirements*
 - This introduces the needs for coverage to ensure the right things have been verified
 - *have been fulfilled*
 - Requires an act of verification to be associated with a coverage measurement

Verification fundamentals



Remember: If you don't look it hasn't been verified
If it hasn't been verified against something objective, then it isn't trustworthy

Only place this happens is in the checker

Fundamentals - Coverage

- The extent or degree to which something is observed, analyzed, and reported.

thefreedictionary.com

- High coverage does not imply high quality
 - With the metrics that are in use today
- While coverage metrics are objective (the information they provide is impassionate)
 - The decision about which to apply is subjective
 - The analysis of results is often subjective
 - Most coverage metrics do not provide “*Confirmation by examination and provisions of objective evidence*”
 - *This include code coverage, functional coverage and almost all other metrics in use*

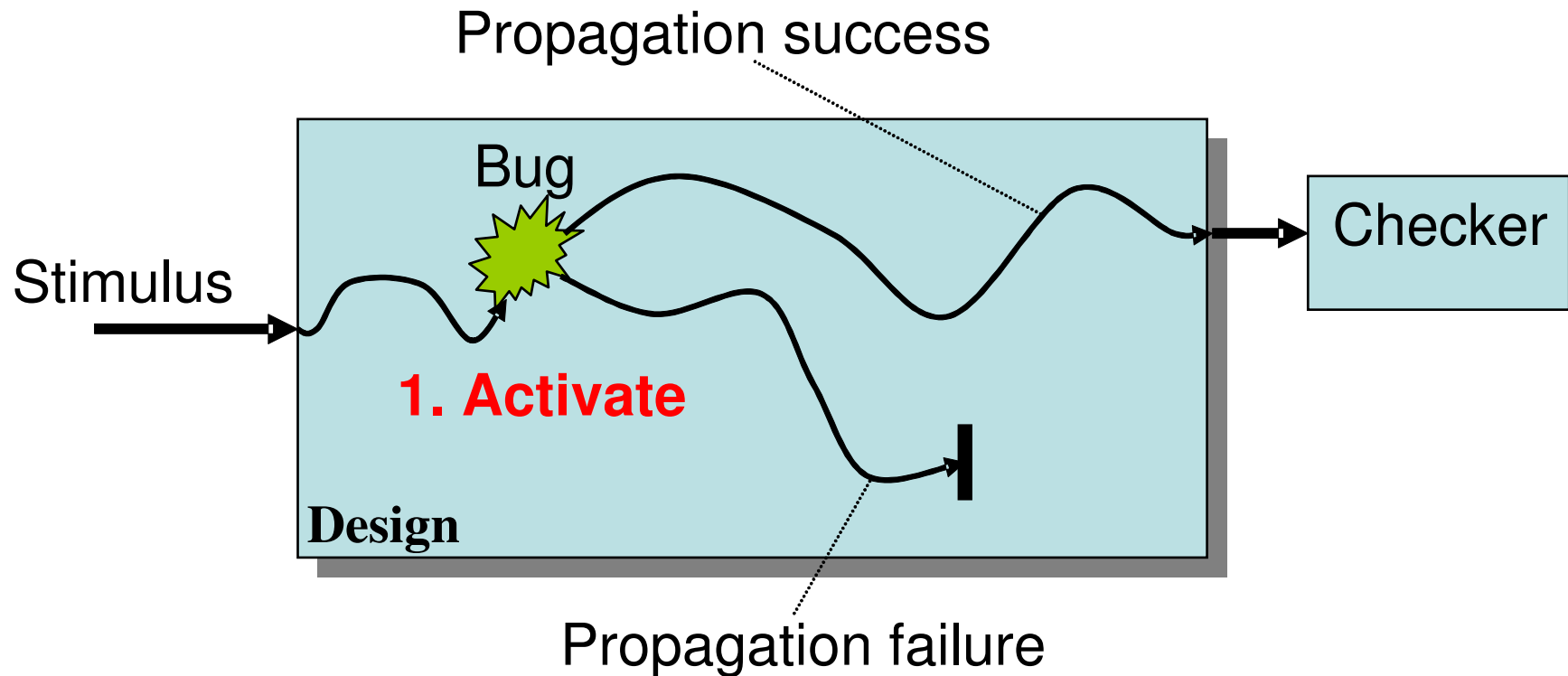
Structural Metrics

- **This is a large class of metrics including**
 - **Code coverage**
 - Line coverage
 - Branch coverage
 - Expression coverage
 - ...
 - **Path coverage**
 - **Toggle coverage**
 - **FSM coverage**
 - State
 - Transition

Structural Metrics

- These metrics are automatically extracted from the implementation source
 - ✓ Easy to implement
 - ✗ Cannot identify what is missing
- They tell you that something was 'reached'
 - A line of code
 - A decision point
 - A state
- ✗ They do not tell you that it was reached for the right reason
- ✗ They do not tell you that the right thing happened after that

Structural Coverage



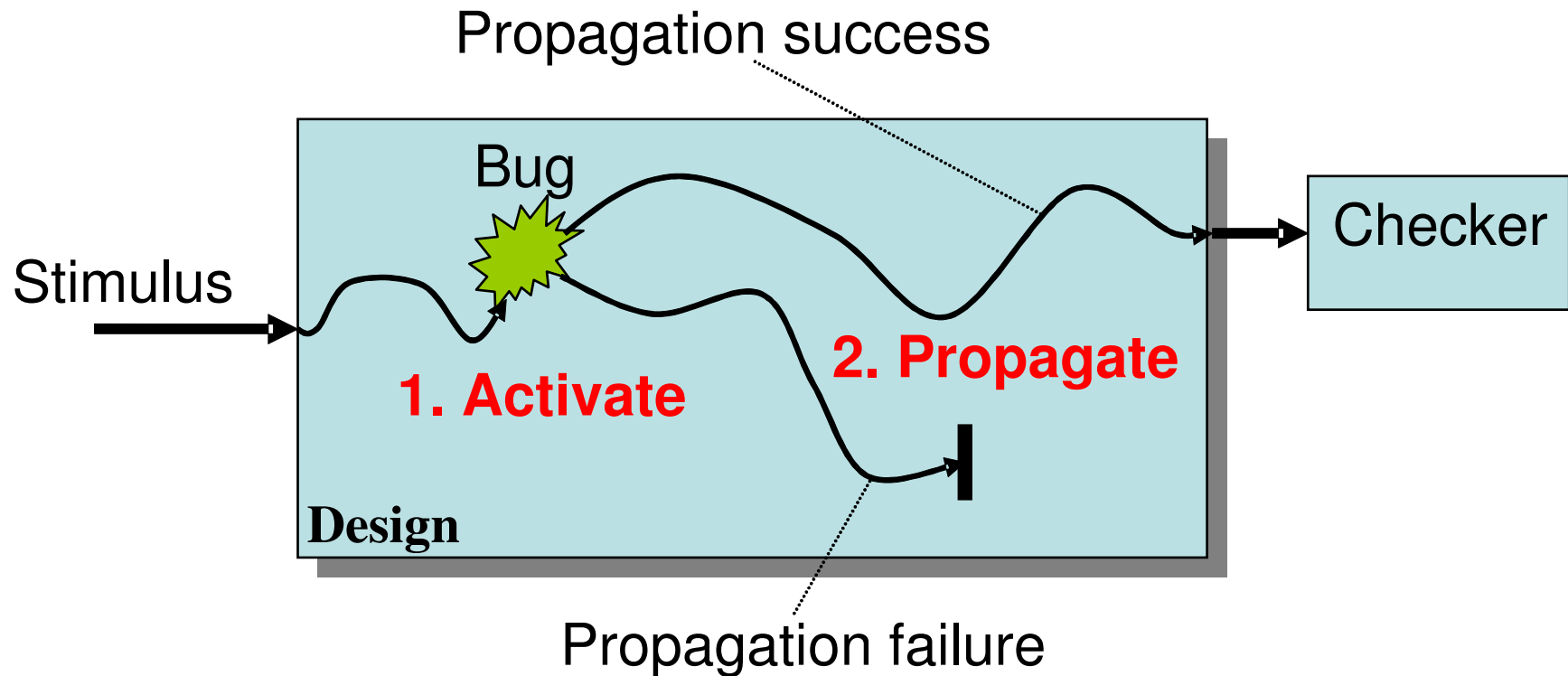
Does not show you that the bug would have been detected where a bug may be

Structural metrics

- **Path coverage**

- ☒ **This is the set of all combinations of all branches**
- ☒ **It identifies exhaustive execution of a model**
- ☒ **Still cannot identify missing functionality**
- ☒ **Expensive computationally**
 - **Limited path sets have been defined but not in use**
- ☒ **Does not identify data errors that do not affect control**
 - **Would not have identified Intel FP bug since this was related to values in a ROM**

Path Coverage



**Does not show you that the
pos bug would have been detected on**

Structural coverage – dirty word

- **When did structural coverage become a dirty word?**
 - Directed tests target specific functionality
 - structural coverage was an impartial way to identify holes
- **Along came constrained/pseudo-random generation**
 - No longer targeting specific functionality
 - Needed a way to ensure important functionality was executed
- **Thus functional coverage was born**
 - There is nothing wrong with structural coverage coupled to directed testing
 - Unless used irresponsibly
 - Becoming less useful with increased concurrency

Functional Coverage

- **Identifies indicators of functionality**
 - **A data value**
 - **A specific state**
 - **A sequence of states**
 - **etc**
- ☒ **It does not identify that the functionality is correct**
 - **Still expects that the comparison of two models is happening**
 - **Suffers from some of the same problems as code coverage**

Functional coverage

- **It is a third independent model**
 - Does not define the correct behaviors, just the behaviors that should be detectable
 - Different language
 - Good and bad
 - ☒ No way to define completion
 - Implementation is subjective
 - Closest theoretical model is path coverage
 - Expensive in terms of execution time
- **Higher cost than structural coverage**
 - ☒ An extra model to define
 - ☒ Slows down simulator
 - ☑ Analysis of holes is easier

Functional Coverage != Verification

Assertions

- **An assertion is the execution of a property in a dynamic verification tool**
- **Properties can be used in both dynamic and static verification flows**
- **A property is a formal description of something that must hold true (or false) in the design**
- **Based on “different” concepts**
 - **They are not procedural languages**
 - **Declarative**
 - ☒ **Takes time to learn and use successfully**
 - ☒ **Allow localized checkers to be placed throughout the implementation**

Assertion Coverage

- **Many different answers:**
 - **How much of the design is 'covered' by assertions**
 - **Assertions per lines of HDL**
 - **Deeper analysis into the points that are touched by assertions**
 - **Were assertions executed**
 - **Was the antecedent executed**
 - **Was the consequent executed**
 - **How many possible paths through the assertion were traversed**
 - **Direct definition of a functional coverage model**
 - **Use of the cover statement on a property**

Summary of coverage methods

	Cost to implement	Cost to execute	Imp or spec	Completion	Objective	Benefits	Analysis
Code	Low	Low	Imp	Low	Yes	Low	Difficult
Path	Low	Low	Imp	High	Yes	High	Moderate
Toggle	Low	Low	Imp	Medium	Yes	Medium	Moderate
Functional	Medium	Medium	Spec	?*	No	Medium	Easy
Assertion	High	High	Spec	?*	No	High	Difficult

*** No direct way to measure today**

Adding Propagation

- **Path Coverage**
 - **Already talked about this**
- **OCCOM** (Observability-based Code Coverage Metrics)
 - Srinivas Devadas Abhijit Ghosh Kurt Keutzer – DAC 1998
 - **Computes the probability that an effect of the fault would be propagated**
 - **During normal simulation – variables are tagged**
 - **Positive and negative tags are used**
 - **In a post processing step, these tags are used to compute the probabilities of propagation**

OCCOM results

Ex	#Tags	Directed				Random			
		#Vec	#Observed Tags	Tag Coverage%	Line Coverage%	#Vec	#Observed Tags	Tag Coverage%	Line Coverage%
mult	44	109	38	86%	100%	25	34	77%	100%
pport	33	18	29	87%	92%	100	22	67%	90%
arbiter	60	88	56	93%	100%	5000	34	57%	90%
count	100	3000	68	68%	92%	3000	68	68%	92%
schsm	52	70	44	85%	94%	4300	28	54%	90%

- **Notes:**

- Shows the problems with coupling code coverage and random techniques
- Shows that people creating directed tests consider propagation
 - This is a flaw with random methods
- Some of the new intelligent testbench tools will make this a lot worse

Propagation – Mutation Analysis

- **Similar to manufacturing test**
 - **Looks for a change in values seen on an output**
 - **Stuck-at faults: what fault model is the equivalent for designer errors?**
 - **Mutation fault model based on two hypotheses:**
 - **that programmers or engineers write code that is close to being correct**
 - **that a test that distinguishes the good version from all its mutants is also sensitive to more complex errors**
 - **Potentially huge number of faults**
- **Concept introduced in 1971**
 - **First tool implementation in 1980**

Mutation Analysis

→ Principles

- Induce mutations (or ***faults***) in the HDL code and show that the verification suite detects those mutations by executing testcases

Original program code:

```
a = b or c;
```

Faulty program code:

```
a = b and c;
```

Typical mutations

- Statement deletion
- Replace each Boolean subexpression with *true* and *false*
- Replace each arithmetic operation with another one, e.g. + with *, - and /
- Replace each Boolean relation with another one, e.g. > with >=, == and <=
- Replace each variable with another variable declared in the same scope (variable types should be the same)

Mutation analysis

- **Performs complete stimulate and propagate analysis**
 - **Addresses --**
 - **If you don't look it hasn't been verified**
 - **But not –**
 - **If it hasn't been verified against something objective, then it isn't trustworthy**
- **This is the same as manufacturing test**
 - **It is not good enough to know that something was different**
 - **Must be able to detect that it was in error**

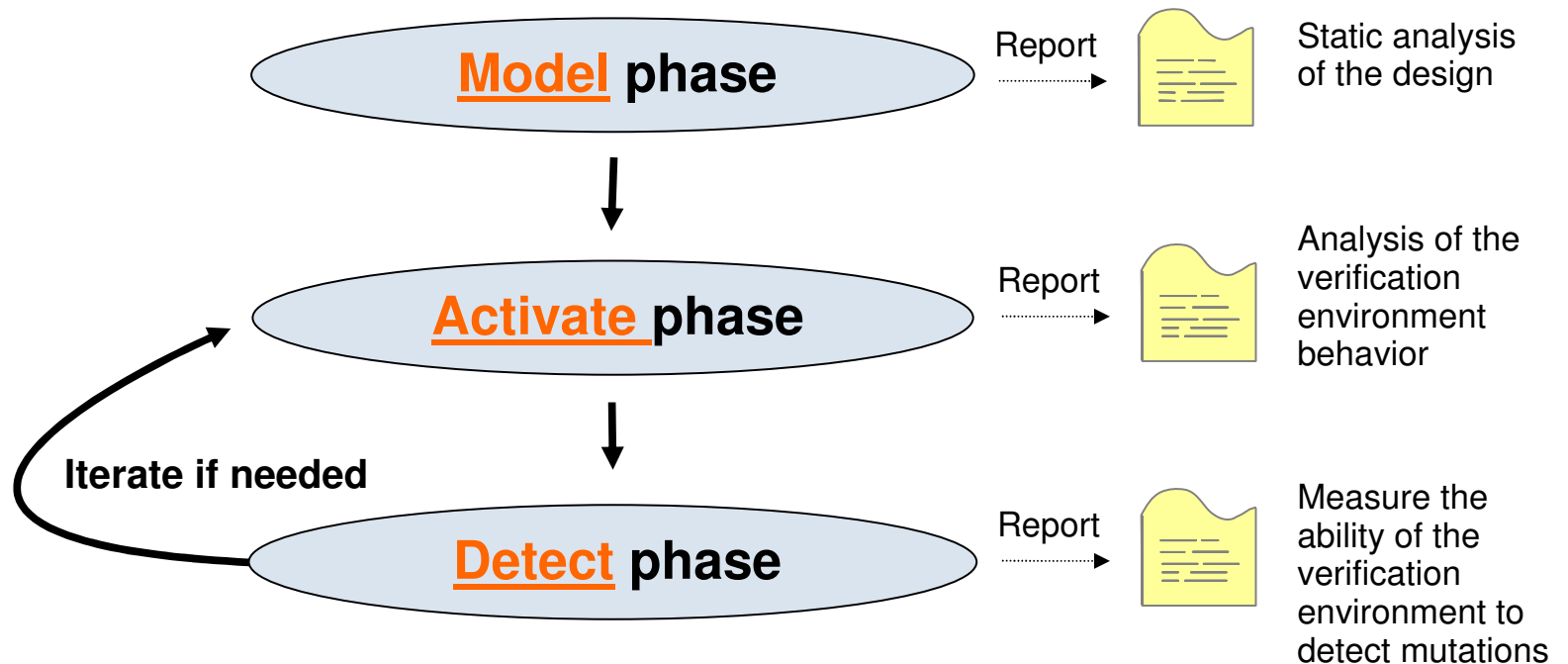
Detection

- **The Holy Grail?**
 - **Can there be a metric or methodology that combines stimulate, propagate and detect?**
 - **Is there a coverage metric that really does measure absolute quality?**
 - **One promising advancement is Functional Qualification from Certess**

Functional Qualification

- **Based on mutation analysis**
- **Several differences:**
 - **Functional Qualification includes the detection phase. For a fault to be *detected* there must be a check made so that at least one testcase fails**
 - **Functional qualification does not depend on propagation to a primary output. Directly supports white box assertions**
 - **Uses very different fault injections schemes to provide relevant results faster**
 - **Applied to hardware instead of software**

Qualification Steps



Qualification Steps

- **Model Phase**
 - **Static analysis of the model to determine the faults to inject**
- **Activate Phase**
 - **Find out which tests activate which faults**
- **Detect Phase**
 - **Perform a run with selected faults injected using selected testcases**

Qualification Flow

The flow resembles:

1. Identify the set of faults to be considered for the mutation analysis
2. Build a meta model of the original program
3. Run each testcase once and observe the internal behavior of the design
4. Compile the meta model program once
5. *Intelligently select a fault not yet analyzed which is unlikely to be detected*
6. Select the subset of testcases that exercise the fault
7. *Intelligently select one testcase not yet run from the subset which is likely to detect*
8. If the testcase fails then goto step 9 else goto step 7
9. *Return to step 5 until some important verification weaknesses are identified*

A small example

RTL

```
module interrupt (input clk,
                 input clear,
                 input result_8,
                 input result_c,
                 input enable,
                 output interrupt);

reg overflow, carry;

always @(posedge clk) begin
    if (clear == 1'b1) begin
        overflow <= 1'b0;
        carry <= 1'b0;
    end
    else begin
        if (result_8 == 1'b1) begin
            overflow <= 1'b1;
        end
        if (result_c == 1'b1) begin
            carry <= 1'b1;
        end
    end
end

assign interrupt = enable ?
    (overflow) : 1'b0;
```

Test

```
"test_interrupt" : begin
    initialize;
    enable = 1;
    // test that result_8 sets the interrupt
    result_8 = 1;
    @(posedge clk);
    check(interrupt,1,"interrupt");
    result_8 = 0;
    // test that result_c sets the interrupt
    result_c = 1;
    @(posedge clk);
    check(interrupt,1,"interrupt");
    result_c = 0;
    pass;
end

assign interrupt = enable ? (overflow) : 1'b0;
```

Checker

```
task check;
    input signal;
    input expect;
    input [800:0] string;
    begin
        if(signal != expect) begin
            $display($time,"ns Error: Expecting %s to be %b but is
            %b",string,expect,signal);
            $finish;
        end
    end
endtask
```

Results

```
35 module interrupt (input clk,
36                   input clear,
37                   input result_8,
38                   input result_c,
39                   input enable,
40                   output interrupt);
41
42 reg overflow, carry;
43
44 always @(posedge clk) begin
45     if (clear == 1'b1) begin
46         overflow <= 1'b0 ;
47         carry <= 1'b0 ;
48     end
49     else begin
50         if (result_8 == 1'b1) begin
51             overflow <= 1'b1 ;
52         end
53         if (result_c == 1'b1) begin
54             carry <= 1'b1 ;
55         end
56     end
57 end
58
59 assign interrupt = enable ? (overflow) : 1'b0 ;
60
```

file:// - Mozilla Firefox

Fault detail

File name: rtl/interrupt.v

Fault ID	Fault type	Status	Detected by	Disabled by Certitude status
17	DeadAssign	Non-Detected		

Affected code:

```
59 assign interrupt = enable ? (overflow) : 1'b0;
```

Is changed into:

```
59 assign /* code removed */;
```

Testcases that activate and propagate the fault:

Testcase	Fault propagated to
test_clear	interrupt.interrupt
test_enable	interrupt.interrupt
test_interrupt	interrupt.interrupt

Report summary:

Explanation of colors is available [here](#).

- 21 faults are injected in the design.
- 0 faults are non-activated.
- 6 faults are non-propagated.
- 14 faults are detected.
- 1 fault is non-detected.
- 0 faults are disabled by Certitude.
- 0 faults are disabled by user.
- 0 faults are dropped.
- 0 faults are not yet qualified.

Fixing the design and testbench

Improved Test

```
"test_interrupt" : begin
  initialize;
  enable = 1;
  // test that result_8 triggers the interrupt
  result_8 = 1;
  @(posedge clk);
  check(interrupt,1,"interrupt");
  result_8 = 0;
  // testcase improvement: clear the interrupt
  // between overflow and
  // carry tests
  // fix ->
  clear = 1;
  @(posedge clk);
  clear = 0;
  // fix <-
  // test that result_c triggers the interrupt
  result_c = 1;
  @(posedge clk);
  check(interrupt,1,"interrupt");
  result_c = 0;
  pass;
end
```

Fix the RTL

**assign interrupt = enable ? (overflow |
carry) : 1'b0;**

Fix the checker

```
if(signal !== expect) begin
  $display($time,"ns Error: Expecting %s to be %b but is  
%b",string,expect,signal);
end
```

Certitude Metrics - ST References

Global Metric

- ◆ Representing the overall quality of the Verification Environment
- ◆ ST reference : 75%, but usually higher

Activation Score

- ◆ Measures the ability of the test suite to exercise all the RTL of the IP
- ◆ Similar to code coverage
- ◆ ST reference : 95%, & 100% explained
- ◆ Missing % should deeply studied & fixed or explained

Propagation Score

- ◆ Measures the ability of the test suite to propagate mutations to the outputs of the IP
- ◆ ST reference : 80%, but should probably be enhanced by adding more test scenarios to reach 90%

Detection Score

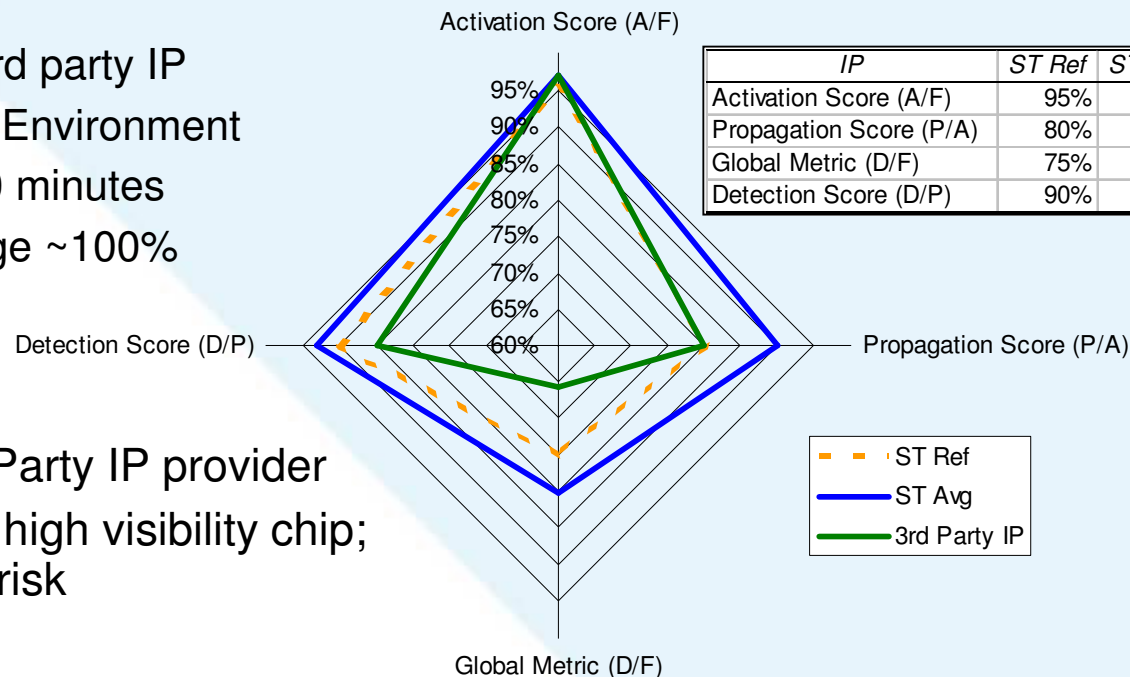
- ◆ Measures the ability of the environment to catch errors
- ◆ ST reference : 90%, but usually higher



Case study 1 : 3rd Party - IP qualification

- **Case study 1:**

- Application: 3rd party IP
- HDL Directed Environment
- ~300 tests, 30 minutes
- Code Coverage ~100%



- **Challenges**

- Convince 3rd Party IP provider
- High revenue, high visibility chip; reduce respin risk

- **Results**

- Helped us to push IP provider to improve verification environment
 - and monitor progress
- Low detection score highlighted manual waveform checks



Applying to other environments

- **This type of analysis measures effectiveness of a testbench**
 - **It is not coverage of a design**
 - **This makes it a fundamental shift from current methods**
- **IBM and at least one provider of formal tools looking at using mutation analysis to measure coverage for formal methods**
- **At least one intelligent testbench provider has used Certitude to identify missing aspects of a graph**

Remaining Issues

- **Performance will always be an issue**
 - It is not clear what the total overhead is
 - ROI may be decisively positive
 - Similar costs to fault simulation
- **Standardized fault model**
 - We need to have multiple tools target the same fault set
- **Detection of missing functionality still an open issue**
 - Less so than with structural coverage alone
- **Best usage model**
 - Should it be used continually or as a final check

Thank You

Questions?

Email: brian_bailey@acm.org

Tel: 503 632 7448

Cell: 503 753 6040

Web: brianbailey.us



HVC'08