COMS30026 Design Verification

Are we there yet?

(The back-end of the verification cycle)

Kerstin Eder

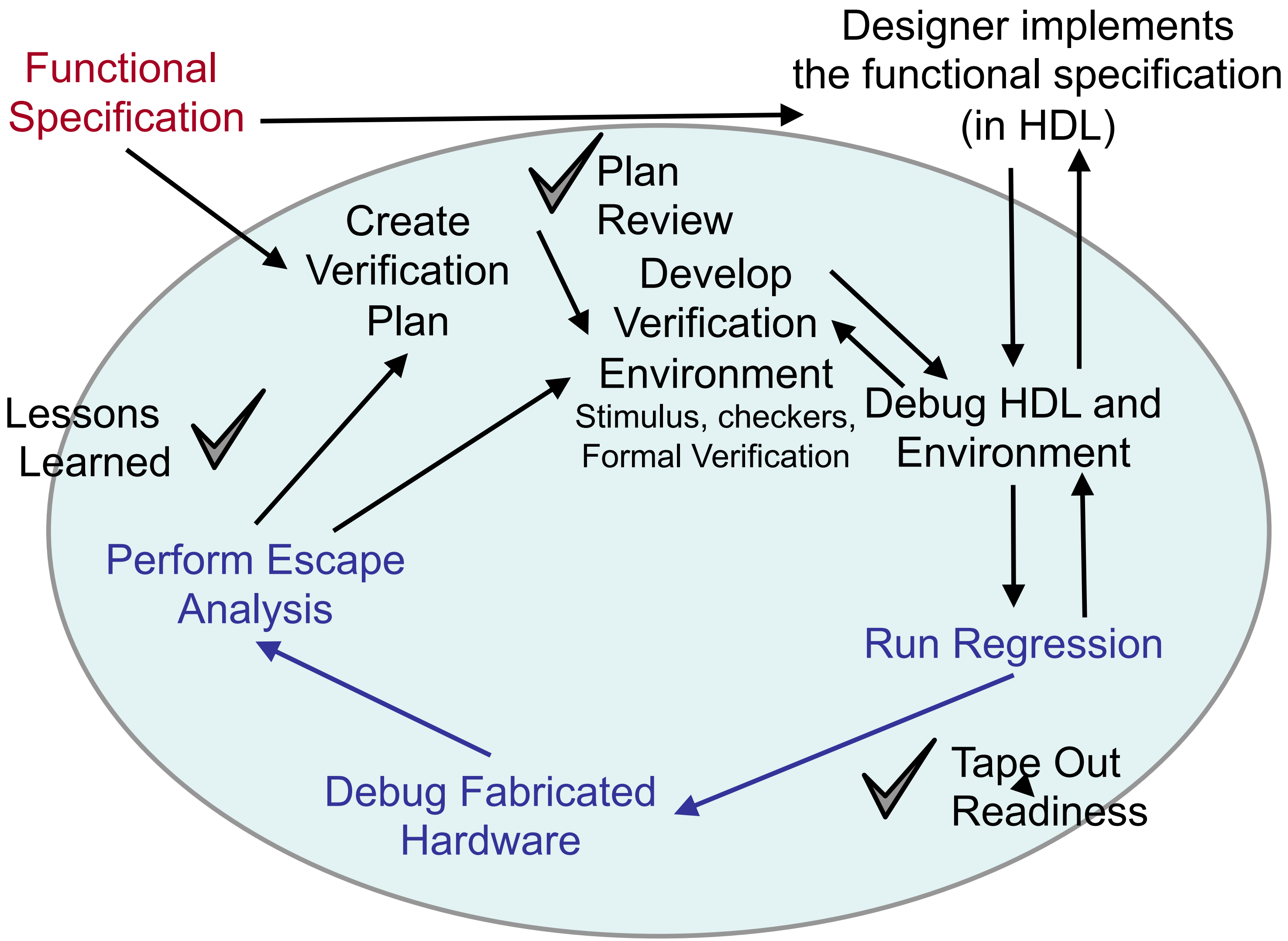
(Acknowledgement: Avi Ziv from the IBM Research Labs in Haifa has kindly permitted the re-use of some of his slides.)

Outline

- The verification cycle - revision
- Coverage closure
- Regression
- Tape-out readiness
- Escape analysis

- Analysis and adaptation
  - Coverage analysis – already covered under “Coverage”
  - Failure analysis – optional material included at end

The Verification Cycle



Functional Specification

Designer implements the functional specification (in HDL)

✓ Plan Review

Create Verification Plan

Develop Verification Environment

Stimulus, checkers, Formal Verification

Debug HDL and Environment

Lessons Learned ✓

Perform Escape Analysis

Run Regression

✓ Tape Out Readiness

Debug Fabricated Hardware

My Environment Is Ready. Now What?

- More functionality was added to the design
  - And therefore, to the verification environment
- Mature enough design is progressed to the next level in the design hierarchy
  - Unit to core to chip to system
- Bugs are being discovered and fixed
  - And bug fixes need to be verified
- The implementation of the verification plan continues
  - Closing holes in coverage
  - Updating the verification plan itself as needed
- Regression is being executed regularly to ensure everything still works

COVERAGE CLOSURE

Coverage Closure

Coverage closure is the process of:

1. Finding areas of coverage not exercised by a set of tests, called Coverage Holes!
2. Creating additional tests to increase coverage by targeting these holes.
   - Beware: Aim to “balance” coverage!

Coverage Closure

Coverage closure is the process of:

1. Finding areas of coverage not exercised by a set of tests, called Coverage Holes!
2. Creating additional tests to increase coverage by targeting these holes.
   - Beware: Aim to “balance” coverage!
   - During coverage closure we may face controllability issues:
     - If the cases to be hit contain DUV internal states/signals (flags), tests that directly exercise all combinations are often hard to find because we can only indirectly control these from the primary inputs of the DUV.

80/20 Split

In practice: 80/20 (20/80) split wrt coverage progress.

Good news:)

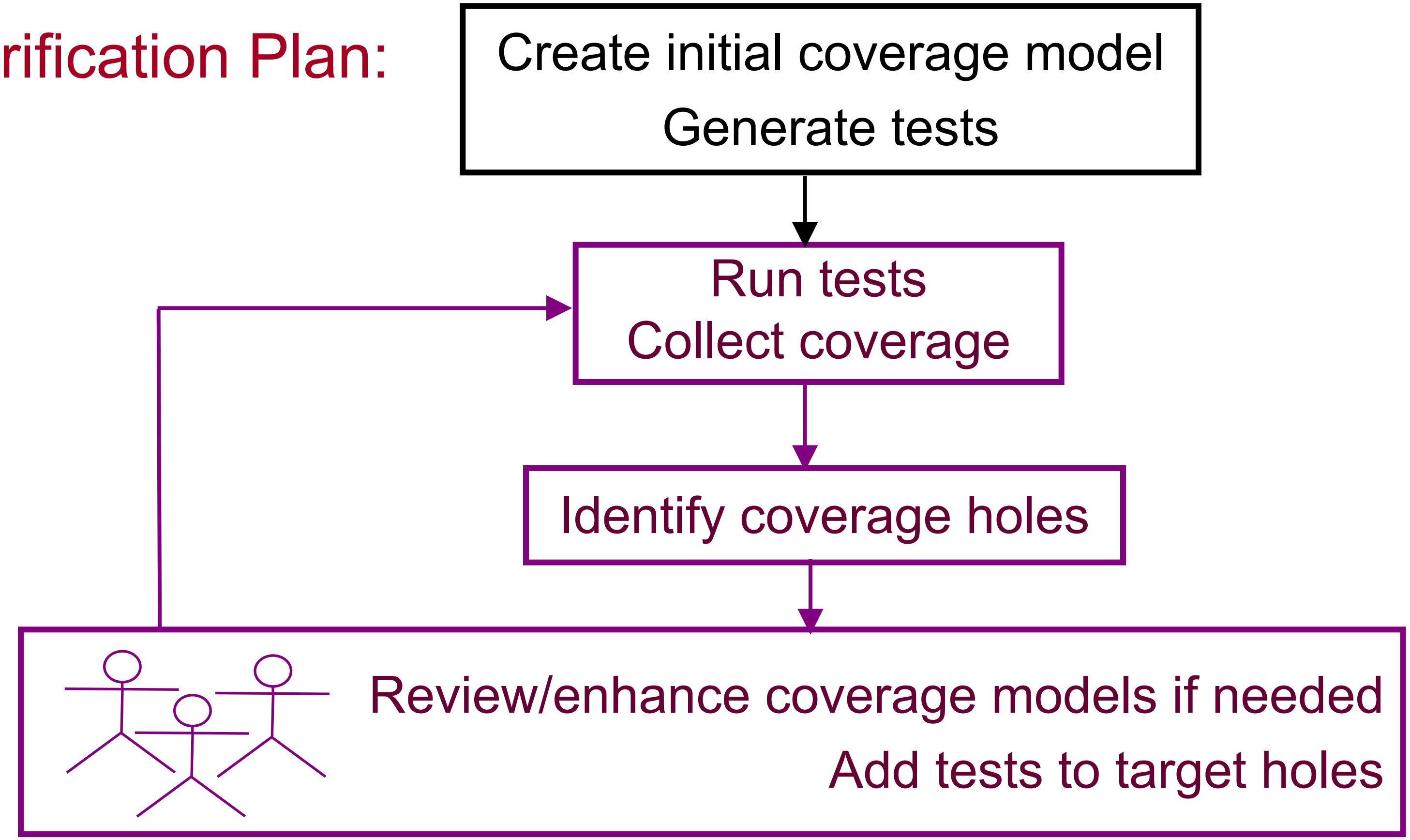
- 80% of coverage is achieved (relatively quickly/easily) driving randomly generated tests.
- This takes about 20% of total time/effort/sim runs spent on verification.

Bad news:(

- Gaining the remaining 20% coverage,
  - i.e. filling the remaining coverage holes (which often needs to be done manually and requires a lot of engineering skill plus design understanding),
- can take as much as 80% of the total time/effort/sim runs spent on verification.

Coverage-Driven Verification Methodology

From Verification Plan:



Create initial coverage model

Generate tests

Run tests

Collect coverage

Identify coverage holes

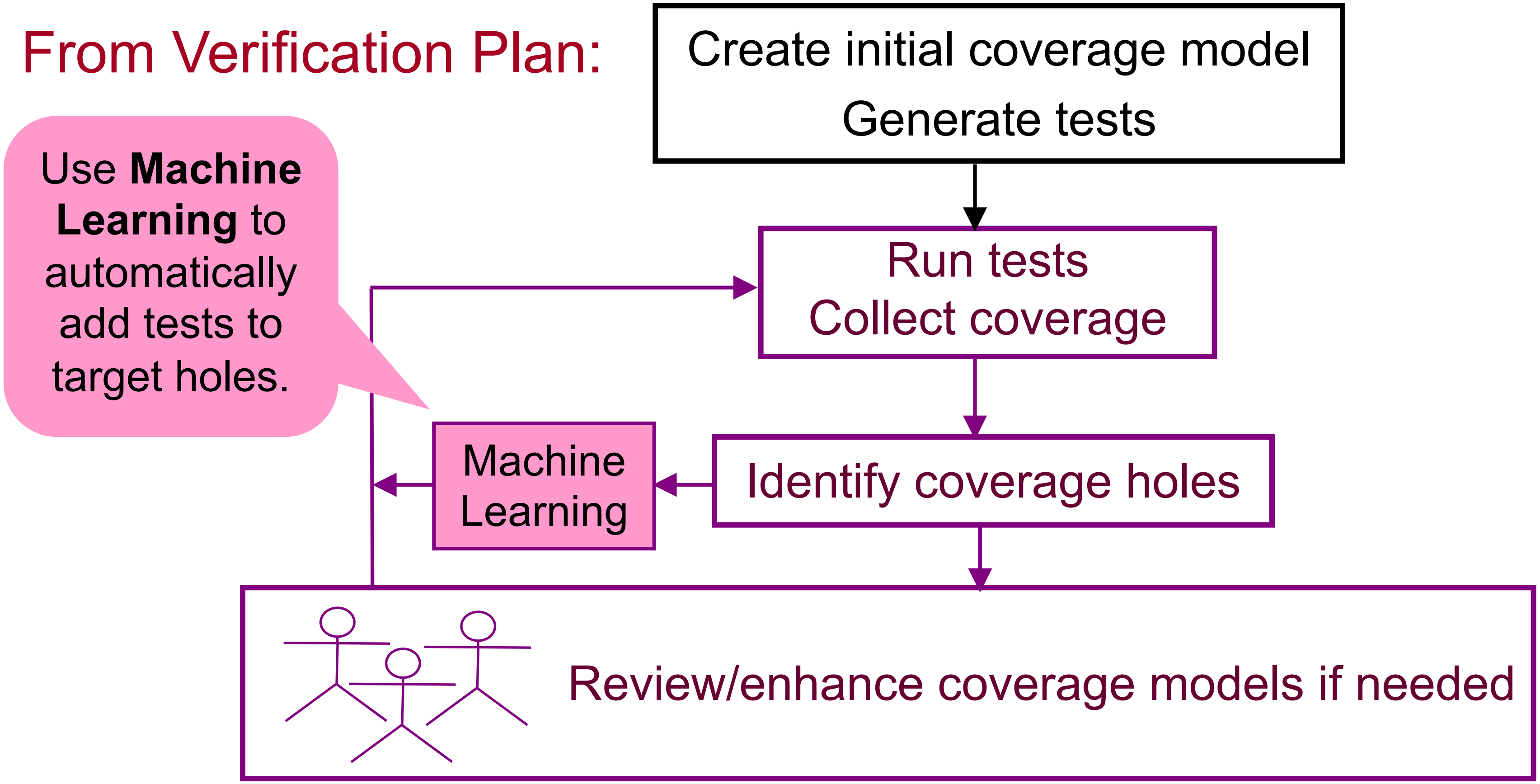
Review/enhance coverage models if needed

Add tests to target holes

Current research: How can we automate this further?

Coverage-DIRECTED Test Generation

From Verification Plan:



Create initial coverage model

Generate tests

Run tests

Collect coverage

Identify coverage holes

Machine Learning

Review/enhance coverage models if needed

Use **Machine Learning** to automatically add tests to target holes.

Current research: How can we automate this further?

CDG: Coverage-DIRECTED Test Generation

How can we make better use of coverage data to **automate** stimulus generation?

Latest Research:

- BY CONSTRUCTION
  - Require description of design as FSM.
  - Use formal methods to derive witness traces.
  - Automatically translate witness traces to test vectors.
  - *Falls over in practice:* FSMs are prohibitively large!

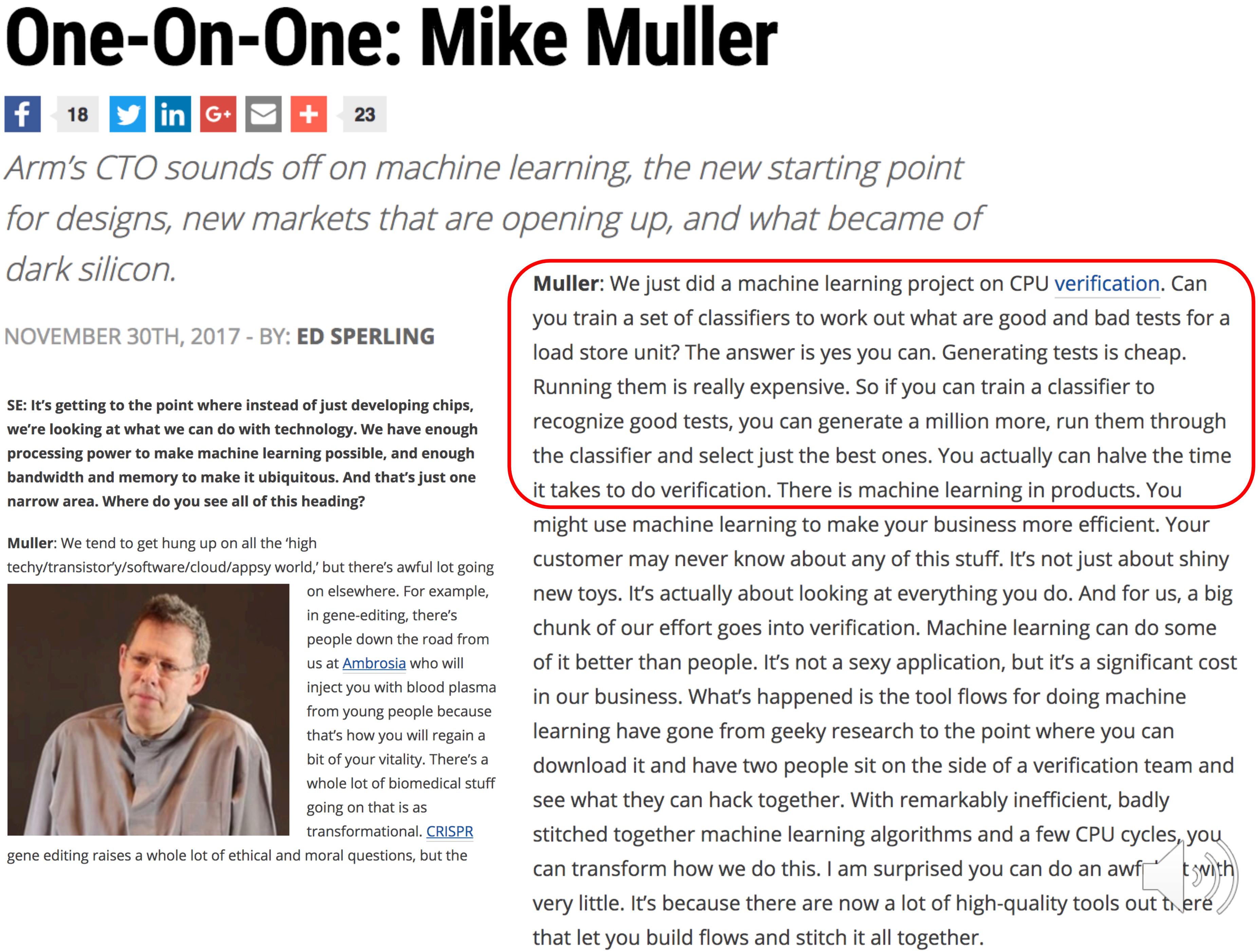
CDG: Coverage-DIRECTED Test Generation

How can we make better use of coverage data to **automate** stimulus generation?

Latest Research:

- BY CONSTRUCTION
  - Require description of design as FSM.
  - Use formal methods to derive witness traces.
  - Automatically translate witness traces to test vectors.
  - *Falls over in practice:* FSMs are prohibitively large!
- BY FEEDBACK

  Exploit Machine Learning techniques
  - GAs/GP - Need to find suitable encoding (e.g. of instructions).
  - Bayesian Networks - Need to design and train BN.
  - Data Mining in coverage spaces – Tend not to scale that well.



One-On-One: Mike Muller

Arm's CTO sounds off on machine learning, the new starting point for designs, new markets that are opening up, and what became of dark silicon.

NOVEMBER 30TH, 2017 - BY: **ED SPERLING**

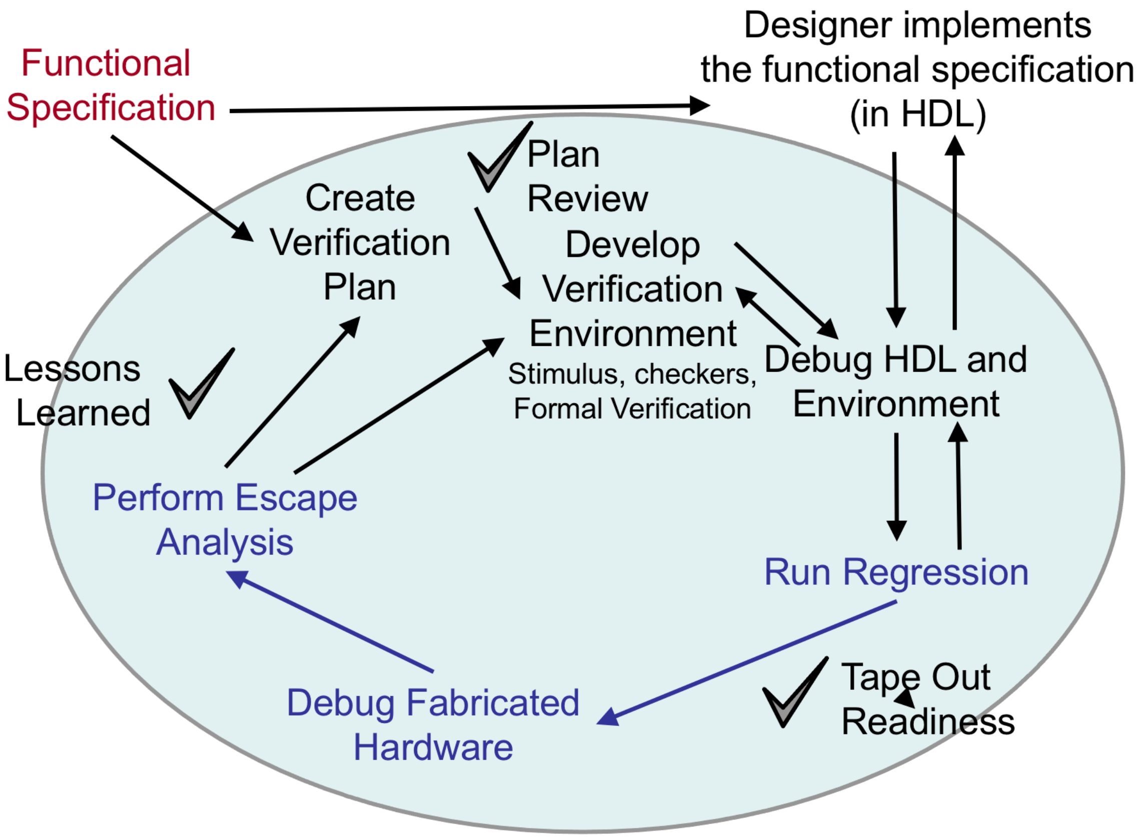
SE: It's getting to the point where instead of just developing chips, we're looking at what we can do with technology. We have enough processing power to make machine learning possible, and enough bandwidth and memory to make it ubiquitous. And that's just one narrow area. Where do you see all of this heading?

Muller: We tend to get hung up on all the 'high techy/transistor'y/software/cloud/appsy world,' but there's awful lot going on elsewhere. For example, in gene-editing, there's people down the road from us at Ambrosia who will inject you with blood plasma from young people because that's how you will regain a bit of your vitality. There's a whole lot of biomedical stuff going on that is as transformational. CRISPR gene editing raises a whole lot of ethical and moral questions, but the

Muller: We just did a machine learning project on CPU verification. Can you train a set of classifiers to work out what are good and bad tests for a load store unit? The answer is yes you can. Generating tests is cheap. Running them is really expensive. So if you can train a classifier to recognize good tests, you can generate a million more, run them through the classifier and select just the best ones. You actually can halve the time it takes to do verification. There is machine learning in products. You might use machine learning to make your business more efficient. Your customer may never know about any of this stuff. It's not just about shiny new toys. It's actually about looking at everything you do. And for us, a big chunk of our effort goes into verification. Machine learning can do some of it better than people. It's not a sexy application, but it's a significant cost in our business. What's happened is the tool flows for doing machine learning have gone from geeky research to the point where you can download it and have two people sit on the side of a verification team and see what they can hack together. With remarkably inefficient, badly stitched together machine learning algorithms and a few CPU cycles, you can transform how we do this. I am surprised you can do an awf t with very little. It's because there are now a lot of high-quality tools out t ere that let you build flows and stitch it all together.

Summary: Coverage Closure

- Verification Methodology should be **coverage-driven**.
  - Shortens implementation time
  - Improves quality
  - Accelerates verification closure
- Need for further automation
  - Research into coverage-directed test generation
- **Delays in coverage closure** are the main reason why verification projects fall behind schedule!


Functional Specification
Designer implements the functional specification (in HDL)
Create Verification Plan
✓ Plan Review
Develop Verification Environment
Stimulus, checkers, Formal Verification
Debug HDL and Environment
Lessons Learned ✓
Perform Escape Analysis
Run Regression
✓ Tape Out Readiness
Debug Fabricated Hardware

REGRESSION

Regression Suites

- A regression suite is a set of tests that are run on the verified design on a regular basis
  - – After major changes
  - – Periodically: Every night or every weekend
- Regression goals
  - – Assuring that things that worked did not stop working
    - This is vital because every bug fix, on average, introduces one fifth of a bug
  - – Detecting “unexpected” bugs

Types of Regression

- Static regression
  - – The regression suite is comprised of a set of “interesting” test patterns
    - Tests that have found bugs in the past
    - Tests that are known to reach corner cases
- Random regression
  - – A.k.a. dynamic or probabilistic regression
  - – The regression suite is comprised of a set of test specifications and an execution policy
    - For example:
      - – execute 100 tests of specification A,
      - – 35 tests of specification B, and
      - – 20 tests of specification C

Static Vs. Random Regression

- Static regression
  - ☑ Known, guaranteed quality
  - ☒ Sensitive to changes
  - ☒ Hard to maintain
- Random regression
  - ☒ Unknown quality
  - ☑ Less sensitive to changes
  - ☑ Easy to maintain
  - ☑ Easy to adapt e.g. to simulation resources
  - ☑ Easy to adjust focus of testing

The Preferred Solution

- Combination of static and random suites
- Small static suite for cases that are hard to recreate
  - Hard to reach corner cases
  - Tests that discovered hard to find bugs
- Random suites for everything else

Regression Suites Requirements

- A regression suite must be:
  - **Comprehensive** so that it is likely to catch all the bugs introduced
  - **Small** so that it can economically be executed many times
- **How can we make our regression suite small and comprehensive?**
- **Solution:** use coverage information
  - Select a set of tests that collectively achieve all the coverage reached so far
  - Select the smallest possible such set

The Set Cover Problem

- Let $S = \{C_1,...,C_n\}$ be the set of coverage tasks
- Let $T = \{T_1,...,T_m\}$ be a set of tests
  - Each test T_i covers the subset $\{C_{i1}, C_{i2}, ...\}$ of the coverage tasks in S
- The **set cover problem:**

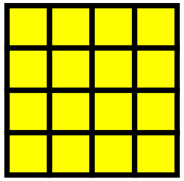
  Find the smallest subset of T that covers S.
- The set cover problem is a known NP-complete problem
  - However, there are several good algorithms for it

Online Algorithm

- For each new test T
  - If T covers an uncovered coverage task
    - Add T to the regression suite

- Advantages
  - Very simple
  - Low memory requirements

Online Algorithm Example



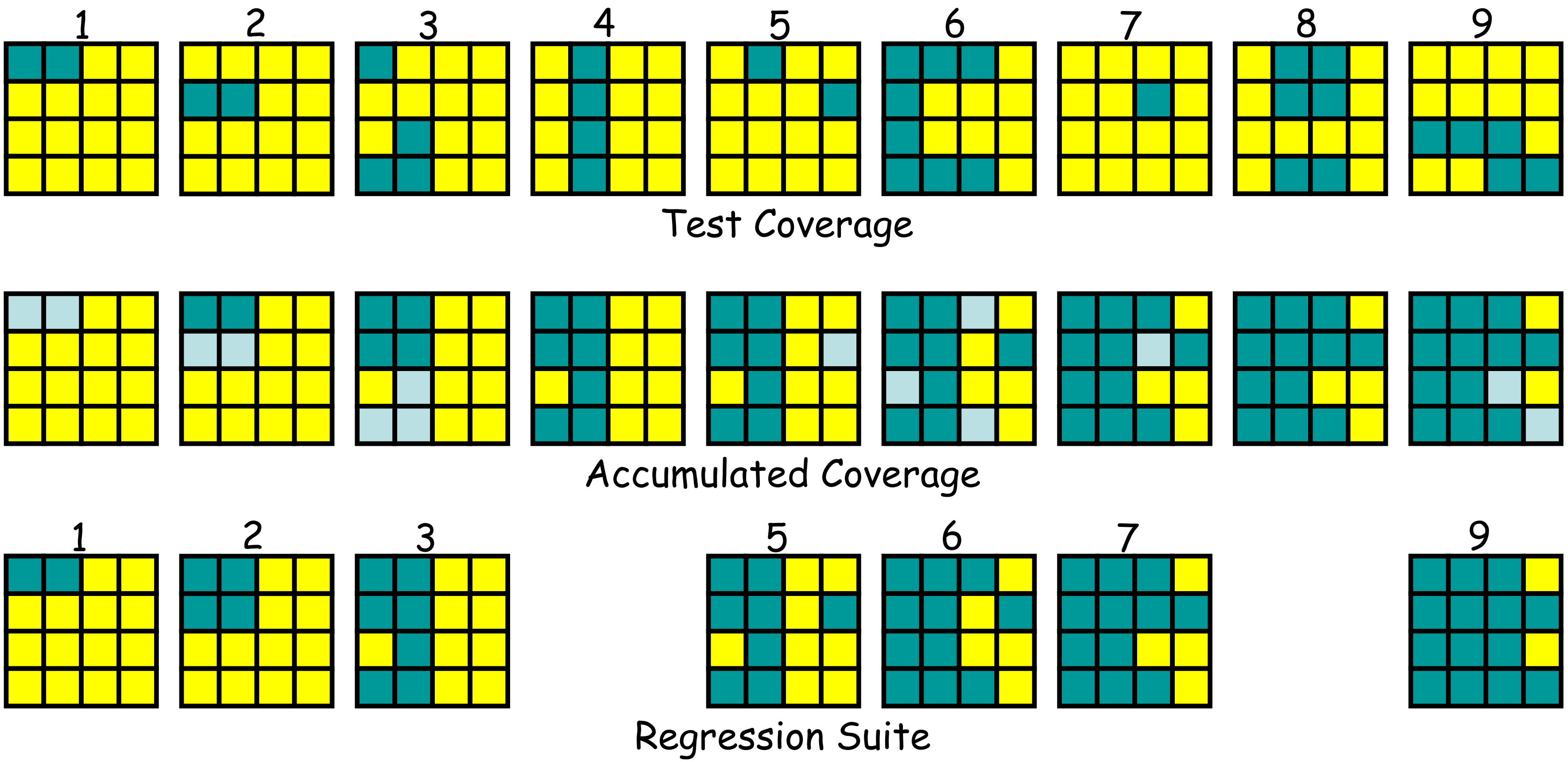
Accumulated Coverage

🟨 Uncovered

🟦 Covered

⬜ Newly covered

Online Algorithm Example



🟨 Uncovered 🟩 Covered ⬜ Newly covered

Greedy Algorithm

- Initialization
  - Build coverage matrix: tests vs. (coverage) tasks
  - Select tests that uniquely cover tasks
- Loop
  - Remove all the tasks covered by selected tests
  - Choose the test that covers most remaining tasks

  until all covered tasks have been addressed

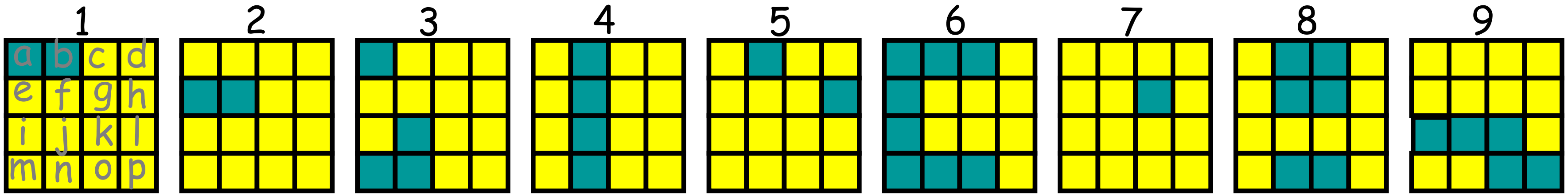
Greedy Algorithm

- Initialization
  - Build coverage matrix: tests vs. (coverage) tasks
  - Select tests that uniquely cover tasks
- Loop
  - Remove all the tasks covered by selected tests
  - Choose the test that covers most remaining tasks

  until all covered tasks have been addressed

- Advantages
  - Quality solution in terms of coverage and size
  - Complexity is polynomial in the number of tests and coverage tasks
- Disadvantage
  - Requires keeping the entire coverage matrix in memory

Greedy Algorithm Example



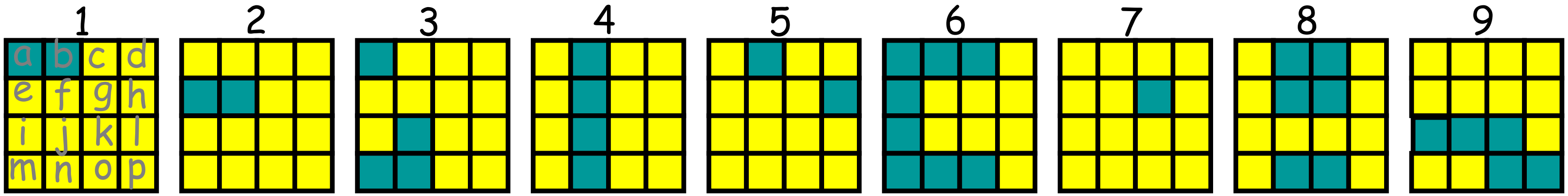
Test Coverage

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	
1	■	■	□	□	□	□	□	□	□	□	□	□	□	□	□	□	2
2	□	□	□	□	■	■	□	□	□	□	□	□	□	□	□	□	2
3	■	□	□	□	□	□	□	□	□	■	□	□	■	■	□	□	4
4	□	■	□	□	□	■	□	□	□	■	□	□	□	■	□	□	4
5	□	■	□	□	□	□	□	■	□	□	□	□	□	□	□	□	2
6	■	■	■	□	■	□	□	□	■	□	□	□	■	■	■	□	8
7	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□	□	1
8	□	■	■	□	□	■	■	□	□	□	□	□	□	■	■	□	6
9	□	□	□	□	□	□	□	□	■	■	■	□	□	□	■	■	5

Number of covered tasks

1. Build Coverage Matrix

Greedy Algorithm Example



Test Coverage

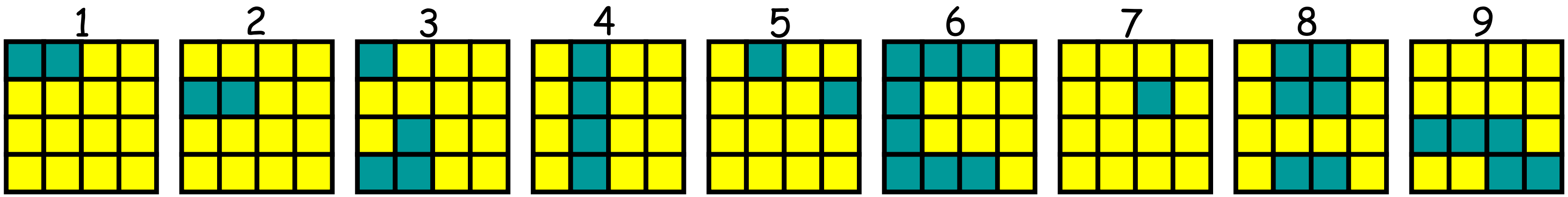
	a	b	c	d	e	f	g	↓ h	i	j	↓ k	l	m	n	o	↓ p	
1	■	■	□	□	□	□	□	□	□	□	□	□	□	□	□	□	2
2	□	□	□	□	■	■	□	□	□	□	□	□	□	□	□	□	2
3	■	□	□	□	□	□	□	□	□	■	□	□	■	■	□	□	4
4	□	■	□	□	□	■	□	□	□	■	□	□	□	■	□	□	4
→ 5	□	■	□	□	□	□	□	■	□	□	□	□	□	□	□	□	2
6	■	■	■	□	■	□	□	□	■	□	□	□	■	■	■	□	8
7	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□	□	1
8	□	■	■	□	□	■	■	□	□	□	□	□	□	■	■	□	6
→ 9	□	□	□	□	□	□	□	□	■	■	■	□	□	□	■	■	5

Number of covered tasks ↑

1. Build Coverage Matrix
2. Select tests that uniquely cover tasks

Regression Suite: 5, 9

Greedy Algorithm Example



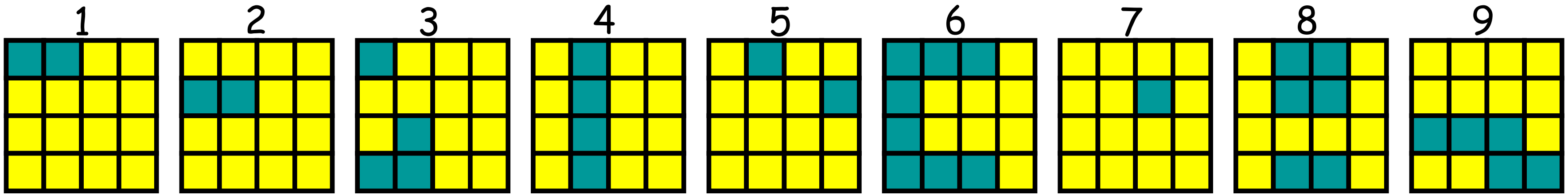
Test Coverage

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	
1	🟦	🟩	🟨	🟨	🟨	🟨	🟨	🟩	🟩	🟩	🟩	🟨	🟨	🟨	🟩	🟩	~~2~~ 1
2	🟨	🟩	🟨	🟨	🟦	🟦	🟨	🟩	🟩	🟩	🟩	🟨	🟨	🟨	🟩	🟩	~~2~~ 2
3	🟦	🟩	🟨	🟨	🟨	🟨	🟨	🟩	🟩	🟩	🟩	🟨	🟦	🟦	🟩	🟩	~~4~~ 3
4	🟨	🟩	🟨	🟨	🟨	🟦	🟨	🟩	🟩	🟩	🟩	🟨	🟨	🟦	🟩	🟩	~~4~~ 2
5	🟨	🟩	🟨	🟨	🟨	🟨	🟨	🟩	🟩	🟩	🟩	🟨	🟨	🟨	🟩	🟩	~~2~~ 0
→ 6	🟦	🟩	🟦	🟨	🟦	🟨	🟨	🟩	🟩	🟩	🟩	🟨	🟦	🟦	🟩	🟩	~~8~~ 5
7	🟨	🟩	🟨	🟨	🟨	🟨	🟦	🟩	🟩	🟩	🟩	🟨	🟨	🟨	🟩	🟩	~~1~~ 1
8	🟨	🟩	🟦	🟨	🟨	🟦	🟦	🟩	🟩	🟩	🟩	🟨	🟨	🟦	🟩	🟩	~~6~~ 4
9	🟨	🟩	🟨	🟨	🟨	🟨	🟨	🟩	🟩	🟩	🟩	🟨	🟨	🟨	🟩	🟩	~~5~~ 0

1. Build Coverage Matrix
2. Select tests that uniquely cover tasks
3. Loop
   a. Remove all the tasks covered by selected tests
   b. Choose the test that covers most remaining tasks

Regression Suite: 5, 9, 6

Greedy Algorithm Example



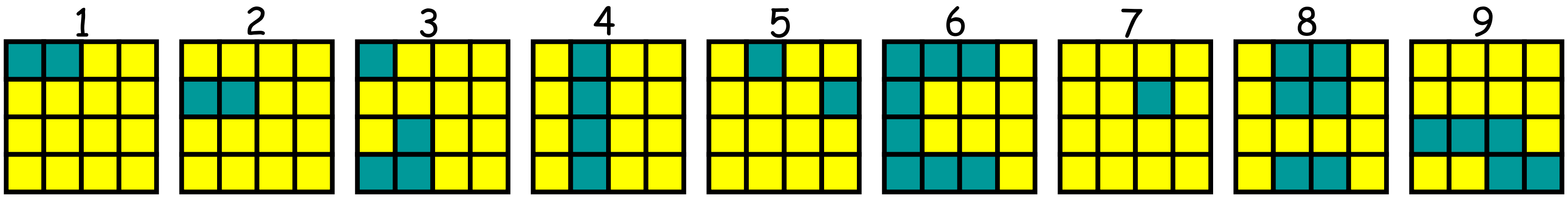
Test Coverage

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	
1	🟩	🟩	🟩	🟨	🟩	🟨	🟨	🟩	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	~~2~~ ~~1~~ 0
2	🟩	🟩	🟩	🟨	🟩	🟦	🟨	🟩	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	~~2~~ ~~2~~ 1
3	🟩	🟩	🟩	🟨	🟩	🟨	🟨	🟩	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	~~4~~ ~~3~~ 0
4	🟩	🟩	🟩	🟨	🟩	🟦	🟨	🟩	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	~~4~~ ~~2~~ 1
5	🟩	🟩	🟩	🟨	🟩	🟨	🟨	🟩	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	~~2~~ ~~0~~ 0
6	🟩	🟩	🟩	🟨	🟩	🟨	🟨	🟩	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	~~8~~ ~~5~~ 0
7	🟩	🟩	🟩	🟨	🟩	🟨	🟦	🟩	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	~~1~~ ~~1~~ 1
→ 8	🟩	🟩	🟩	🟨	🟩	🟦	🟦	🟩	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	~~6~~ ~~4~~ 2
9	🟩	🟩	🟩	🟨	🟩	🟨	🟨	🟩	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	~~5~~ ~~0~~ 0

1. Build Coverage Matrix
2. Select tests that uniquely cover tasks
3. Loop
   a. Remove all the tasks covered by selected tests
   b. Choose the test that covers most remaining tasks

Regression Suite: 5, 9, 6, 8

Greedy Algorithm Example



Test Coverage

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	
1	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	~~2~~ ~~1~~ 0
2	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	~~2~~ ~~2~~ 1
3	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	~~4~~ ~~3~~ 0
4	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	~~4~~ ~~2~~ 1
5	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	~~2~~ ~~0~~ 0
6	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	~~8~~ ~~5~~ 0
7	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	~~1~~ ~~1~~ 1
→ 8	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	~~6~~ ~~4~~ 2
9	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟨	🟩	🟩	🟩	🟩	~~5~~ ~~0~~ 0

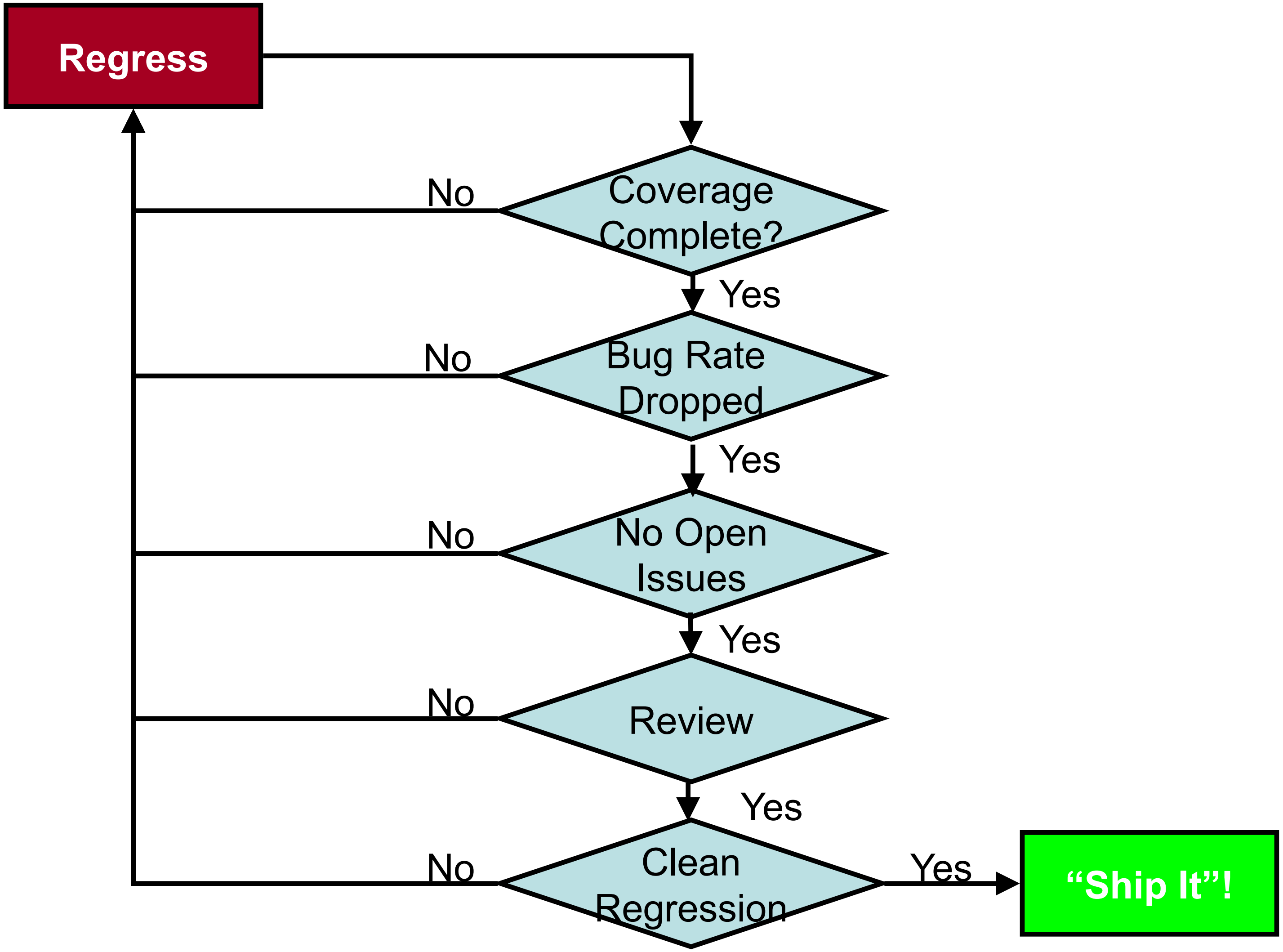
1. Build Coverage Matrix
2. Select tests that uniquely cover tasks
3. Loop
   a. Remove all the tasks covered by selected tests
   b. Choose the test that covers most remaining tasks

Regression Suite: 5, 9, 6, 8

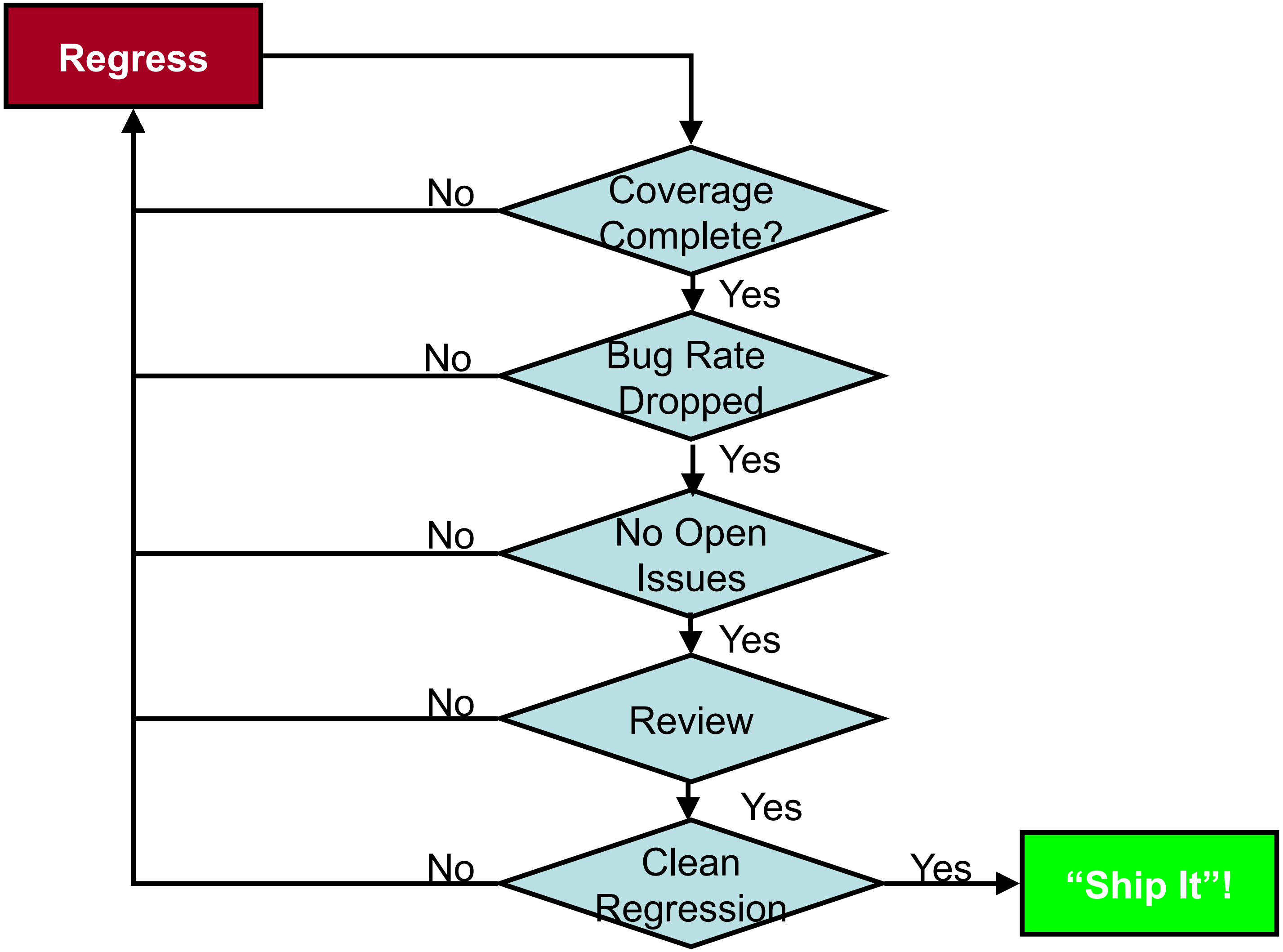
Online Algorithm Regression Suite: 1,2,3,5,6,7,9

COMPLETION CRITERIA

When Is Verification Done?

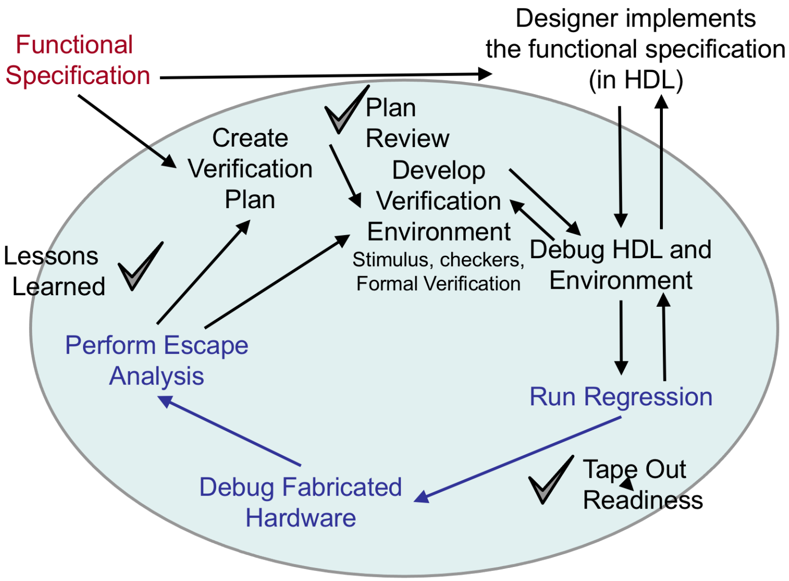


When Is Verification Done?



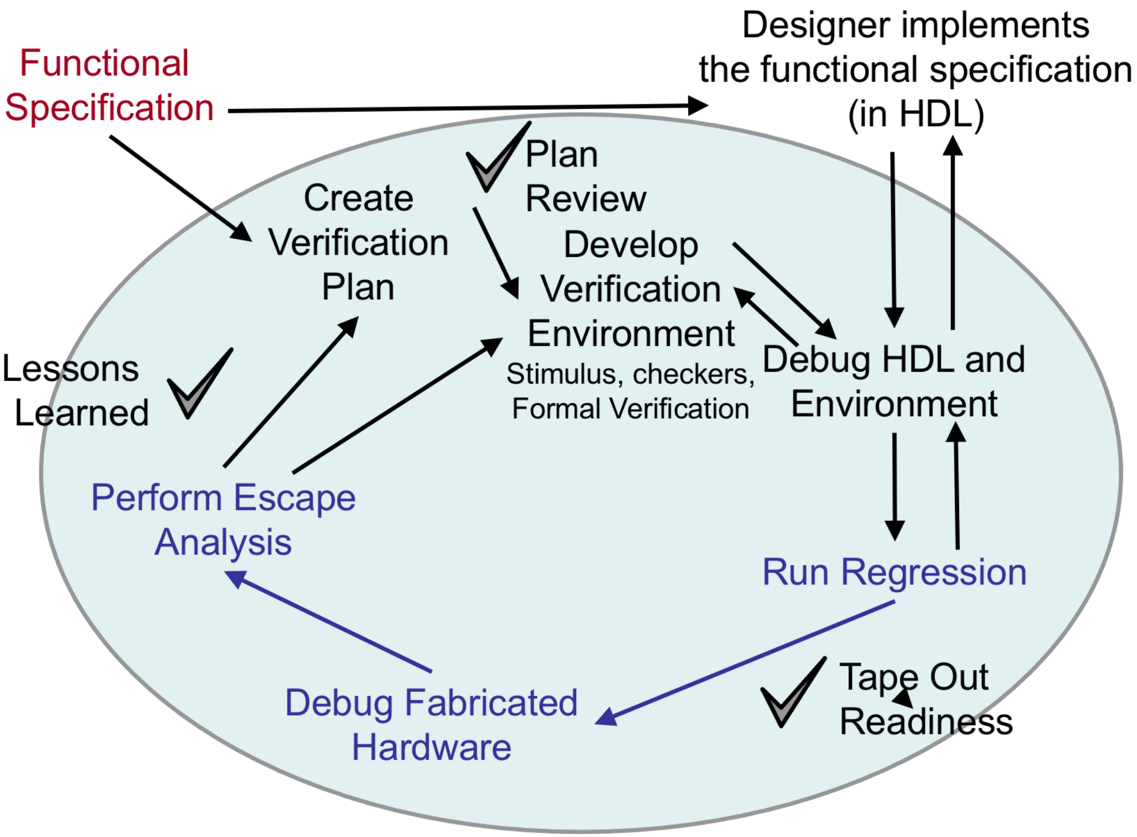
Tape-Out Readiness

- Before sending a design to manufacturing, it must meet established tape-out criteria



Functional Specification
Designer implements the functional specification (in HDL)
Plan Review
Create Verification Plan
Develop Verification Environment
Stimulus, checkers, Formal Verification
Debug HDL and Environment
Lessons Learned
Perform Escape Analysis
Run Regression
Tape Out Readiness
Debug Fabricated Hardware

- The criteria are a series of checklists that indicate completion of planned work
- Verification is just one element in this series of checklists
- Tape-out readiness is measured by a set of metrics
- The most relevant metrics for verification are bug rates and coverage


Functional Specification
Designer implements the functional specification (in HDL)
Plan Review
Create Verification Plan
Develop Verification Environment
Stimulus, checkers, Formal Verification
Debug HDL and Environment
Lessons Learned
Perform Escape Analysis
Run Regression
Tape Out Readiness
Debug Fabricated Hardware

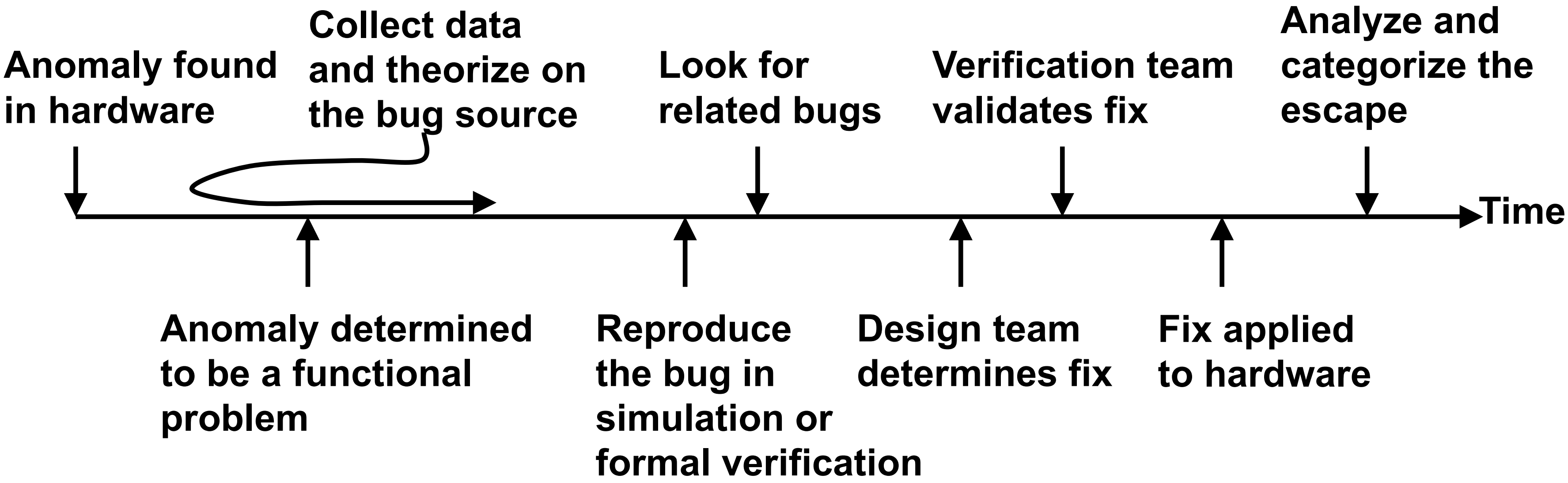
ESCAPE ANALYSIS

Escape Analysis

- An **escape** is a bug found later in the verification process than it should have been
  - In other words, it escaped its target place
  - Usually, escapes refer to bugs found in the hardware itself instead of during simulation

- Escape analysis has two important aspects
  - Make sure that the bug is fully understood and fixed correctly
    - We do not want another tape-out because of a bad fix
  - Understand why the bug escaped simulation in the first place
    - replicate the bug in simulation and
    - improve the verification plan and process to avoid such escapes in the future

Individual Escape Analysis Timeline



Anomaly found in hardware

Anomaly determined to be a functional problem

Collect data and theorize on the bug source

Reproduce the bug in simulation or formal verification

Look for related bugs

Design team determines fix

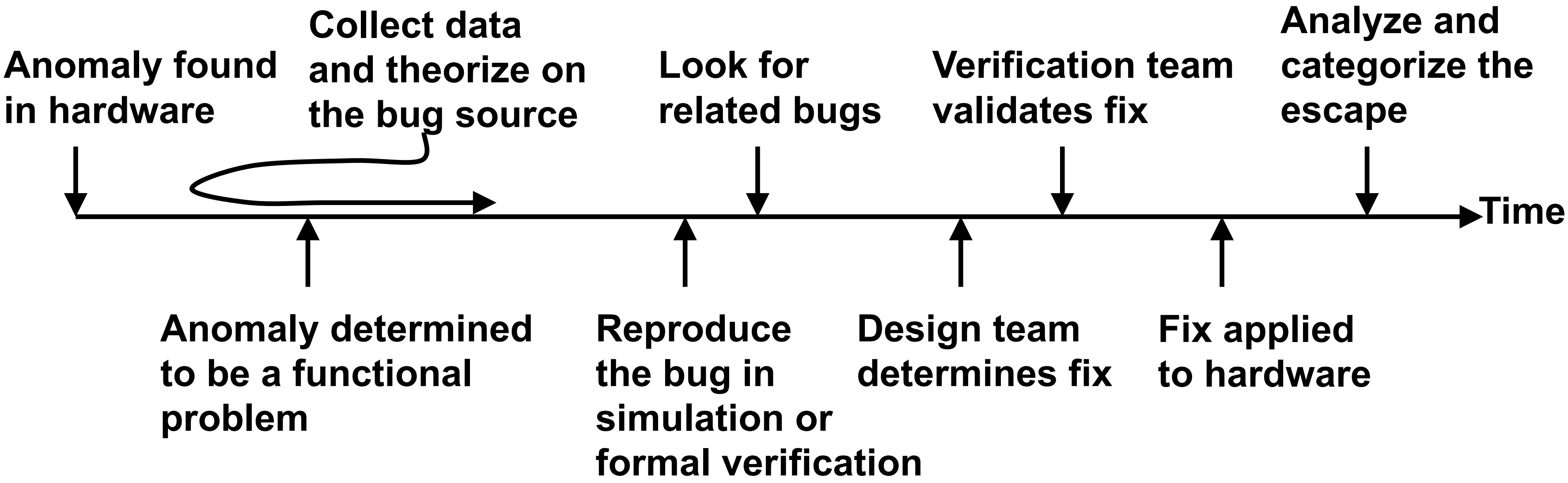
Verification team validates fix

Fix applied to hardware

Analyze and categorize the escape

Time

Individual Escape Analysis Timeline



Anomaly found in hardware

Anomaly determined to be a functional problem

Collect data and theorize on the bug source

Reproduce the bug in simulation or formal verification

Look for related bugs

Design team determines fix

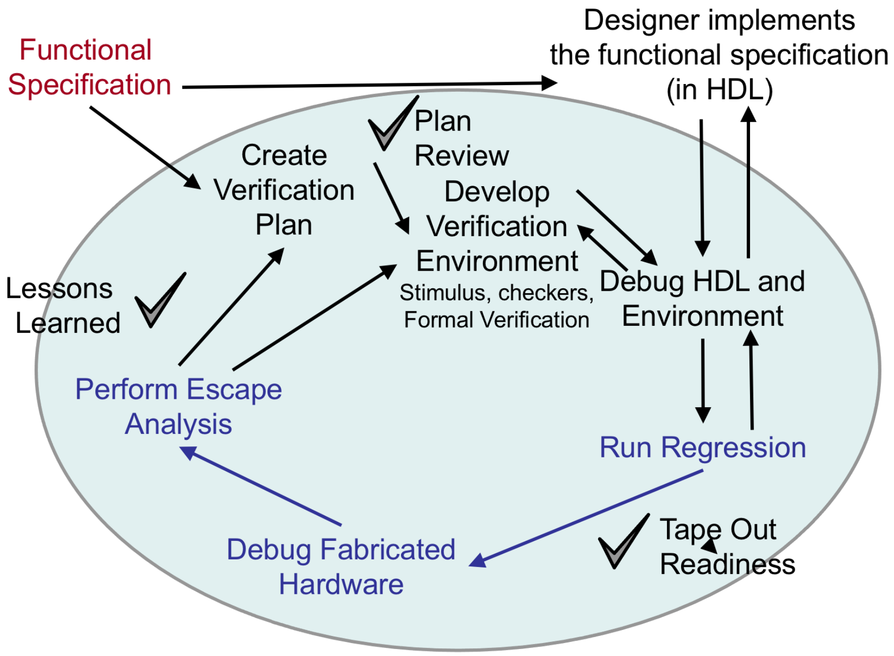
Verification team validates fix

Fix applied to hardware

Analyze and categorize the escape

Summary

- Completion of the Verification Cycle includes:
  - – Coverage closure
  - – Coverage analysis
    - (already under “Coverage”)
  - – Failure analysis*
    - (optional – see attached slides)
  - – Regression
  - – Tape-out readiness
  - – Escape analysis


Functional Specification
Designer implements the functional specification (in HDL)
Plan
Review
Create Verification Plan
Develop Verification Environment
Stimulus, checkers, Formal Verification
Debug HDL and Environment
Lessons Learned
Perform Escape Analysis
Run Regression
Tape Out Readiness
Debug Fabricated Hardware

* Optional material, see attached slides with detailed notes but no narration.

COMS30026 Design Verification

Are we there yet?

(The back-end of the verification cycle)

Analysis and Adaptation

Kerstin Eder

(Acknowledgement: Avi Ziv from the IBM Research Labs in Haifa has kindly permitted the re-use of some of his slides.)

Analysis and Adaptation

- Building a good verification plan is the first step for successful verification
  - But it is not enough!
- Need to constantly:
  - Monitor the verification process
  - Analyze the observations
  - Adapt to address issues identified by the analysis
- Three basic levels of adaptation
  - Change the way the verification environment is activated
  - Change the verification environment
  - Change the verification plan

Two Types of Analysis

1. Coverage analysis
   - Was included in the lectures on coverage.

2. Failure analysis

FAILURE ANALYSIS

Failure Analysis

- During execution of the verification plan (many) failures are observed
- This is not a bad phenomenon
  - Remember that the goal of the verification process is to identify faults in the DUV
- The goal of failure analysis is to understand failures, their causes, their relation to one another, and their relation to the verification process

Failures and Faults

- **Failure** – an observed DUV behavior that violates the specified behavior
- **Fault** – the root cause of a failure
- There can be a many-to-many relationship between faults and failures
  - Mishandling of overflow in the input FIFO can cause:
    - Lost commands in the output port
    - Bad data in the output port
  - Bad data in the output port can be caused by:
    - Mishandling of overflow in the input FIFO
    - Bad selection in the output selector

How Failures Are Detected

- Inspection and code review
- Output of formal verification tools or other static analysis tools, such as lint
- Activation of response checkers during simulation
- Analysis of coverage data
- Visual observation of application misbehavior

Types of Failure Analysis

- Detailed failure analysis
  - Understand the cause and effects of failures and faults on the design, environment, verification process and more
- Statistical failure analysis
  - Identify trends, provide prediction

Detailed Failure Analysis

- The outcome of the analysis
  - The failure is understood and recorded
  - The failure is resolved
  - The verification plan and process are adapted
  - Lessons learned for the future
- Note: In most cases failure analysis—and especially the last two items—are simple and the outcome of the analysis is that we found a failure and a fault when and where expected and because we are doing our job the right way.

Understanding the Failure

- The goal is to understand the scope and severity of the failure and how the failure can be recreated
- Provides useful information for debugging and other parts of the failure analysis
  - Simplify and generalize the failure conditions
    - Find simper settings / stimuli that recreate the failure
    - Find necessary and sufficient conditions for the failure
  - Localize the fault in terms of place and time
  - Research: Generate easy-to-debug tests

What to look for

- In simulation
  - Determinism
    - Does the failure always occur in the same settings?
      - With the same seed?
      - With different seeds (or random seed)?
  - Parameters that are correlated with the failure
    - Parameters that cause the failure to disappear
    - Parameters that cause the failure to change
  - Specific parts in the stimuli that are correlated to the failure
- In formal verification
  - Constraints that affect the failure
  - Time bounds that affect the failure

Resolving the Failure

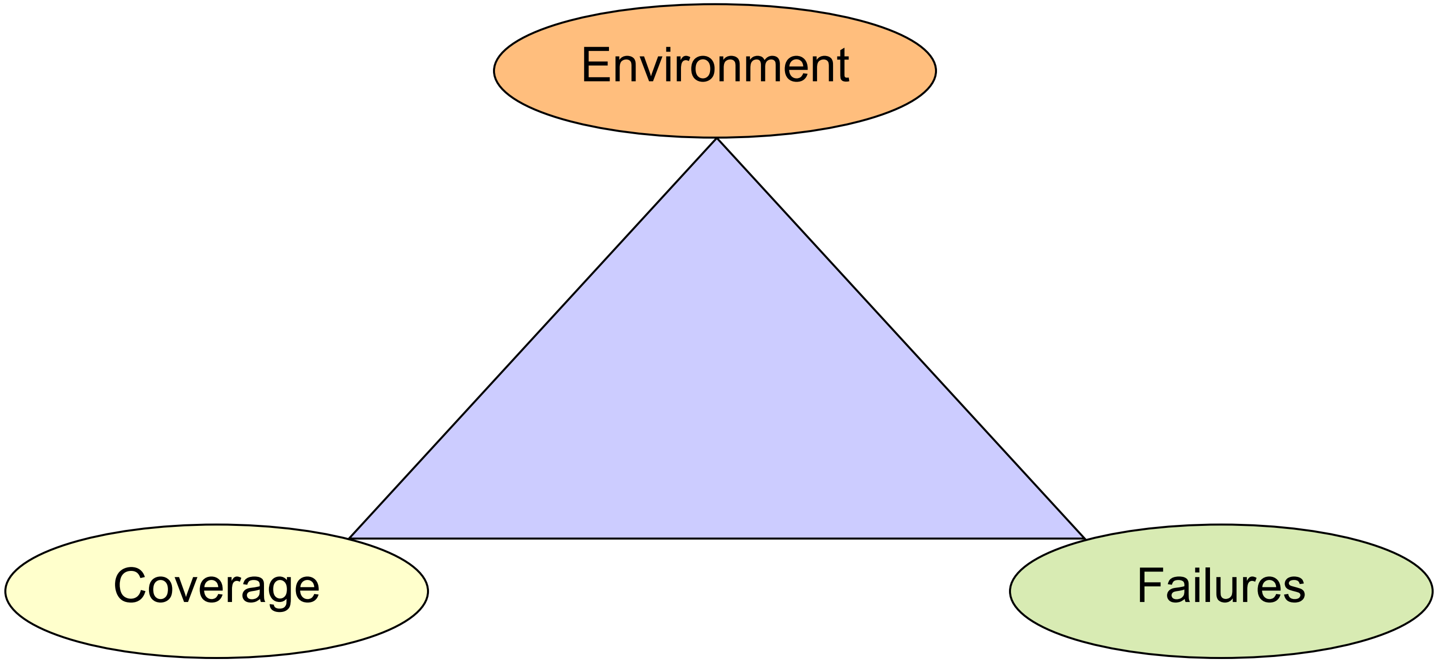
- This does not always mean fixing the fault
  - Defer to future tape outs / releases
  - Bypass by software or surrounding modules
  - Record in errata sheets
- Need to ensure that the resolution is complete
  - The fix / bypass is correct
  - All cases are covered
  - No new faults introduced in the process
  - (Similar cases are also handled)
- Mini-verification plan is needed
  - Coverage models
  - Stimuli generation strategy
  - New result checkers

Adapting the Verification Plan and Process

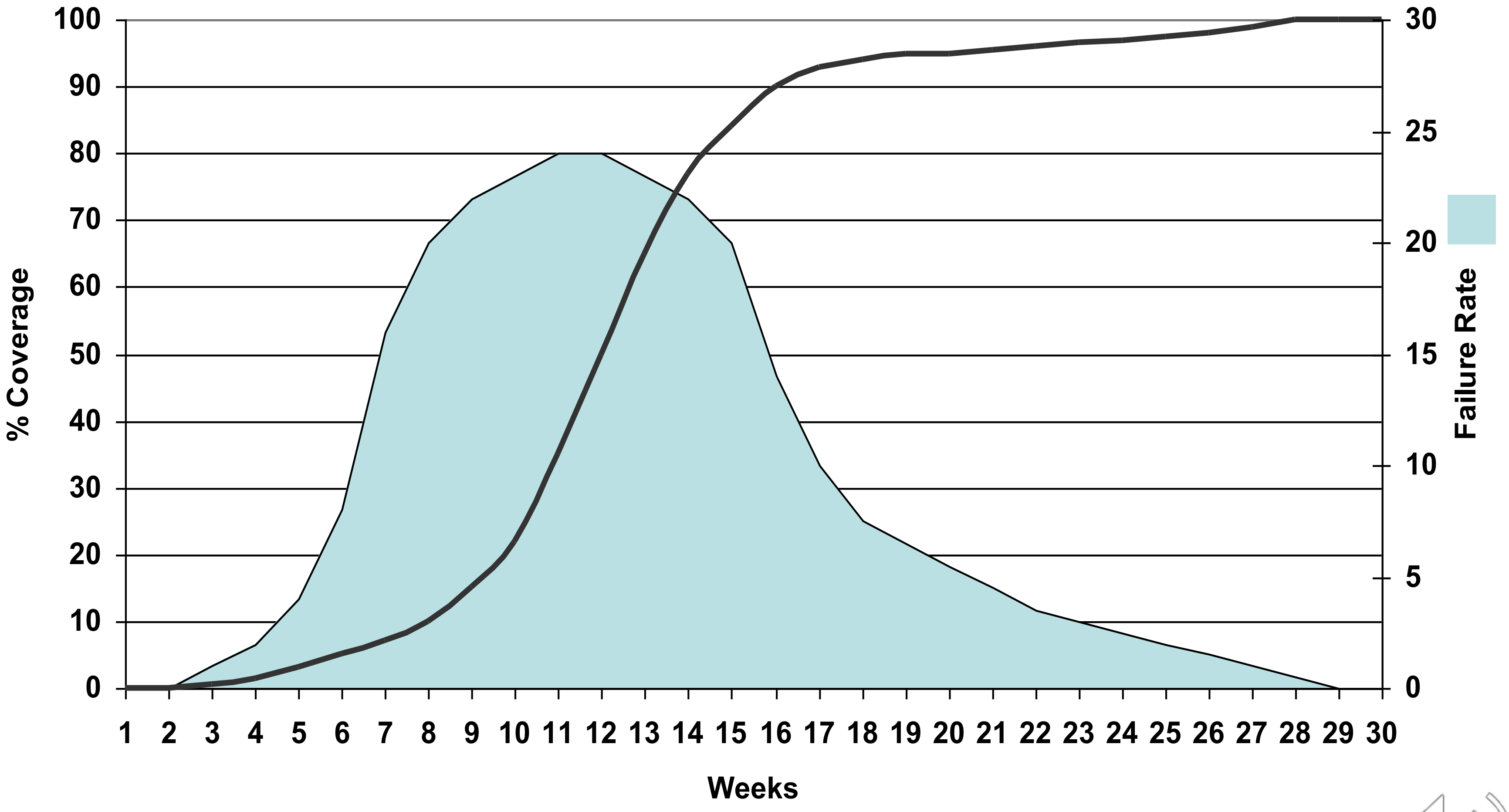
- Need to minimize faults found by chance or found too late
  - These faults can easily be missed if we are less lucky
- Indicators that faults are found by chance
  - Faults are not found at the right time
    - Fault is found at the wrong level of the hierarchy
    - Faults are found not at the area we concentrate on
    - Need to understand why faults are not found at the right time
      - And, change the plan and process accordingly
  - Faults are not found by the right checker
    - Only a side effect of the fault is detected
    - May indicate missing checker or problems in existing checker
  - Simulation with failure is not flagged by coverage
    - Does not activate uncovered or rarely covered coverage points
    - Indicates missing coverage models

Correlating Coverage and Failures

- There is a direct correlation between
  - Changes in the verification environment and the DUV
  - Progress in coverage
  - Detection of new failures



Correlating Failure Rate and Coverage Progress



Individual Coverage and Failure Correlation

- Correlating a failure to specific coverage can be helpful in the failure analysis and debugging processes
- Rare coverage points exercised by a simulation that fails can hint at the location of the fault that caused the failure
  - Rare coverage points are coverage points rarely, if ever, exercised by passing simulations
  - These coverage points record what happened in the DUV prior to the failure
  - They are very useful if the failure is distant (in logic or time) from the fault or the fault is complex
- If no such rare coverage points are recorded, then it is likely that the failure is found by chance
  - The verification plan needs to be refined to catch these failures