

# COMS31700 Design Verification: Coverage

Kerstin Eder

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

# Last Time

---

- Verification Cycle
- Verification Methodology &
- Verification Plan

Previously: **Verification Tools**

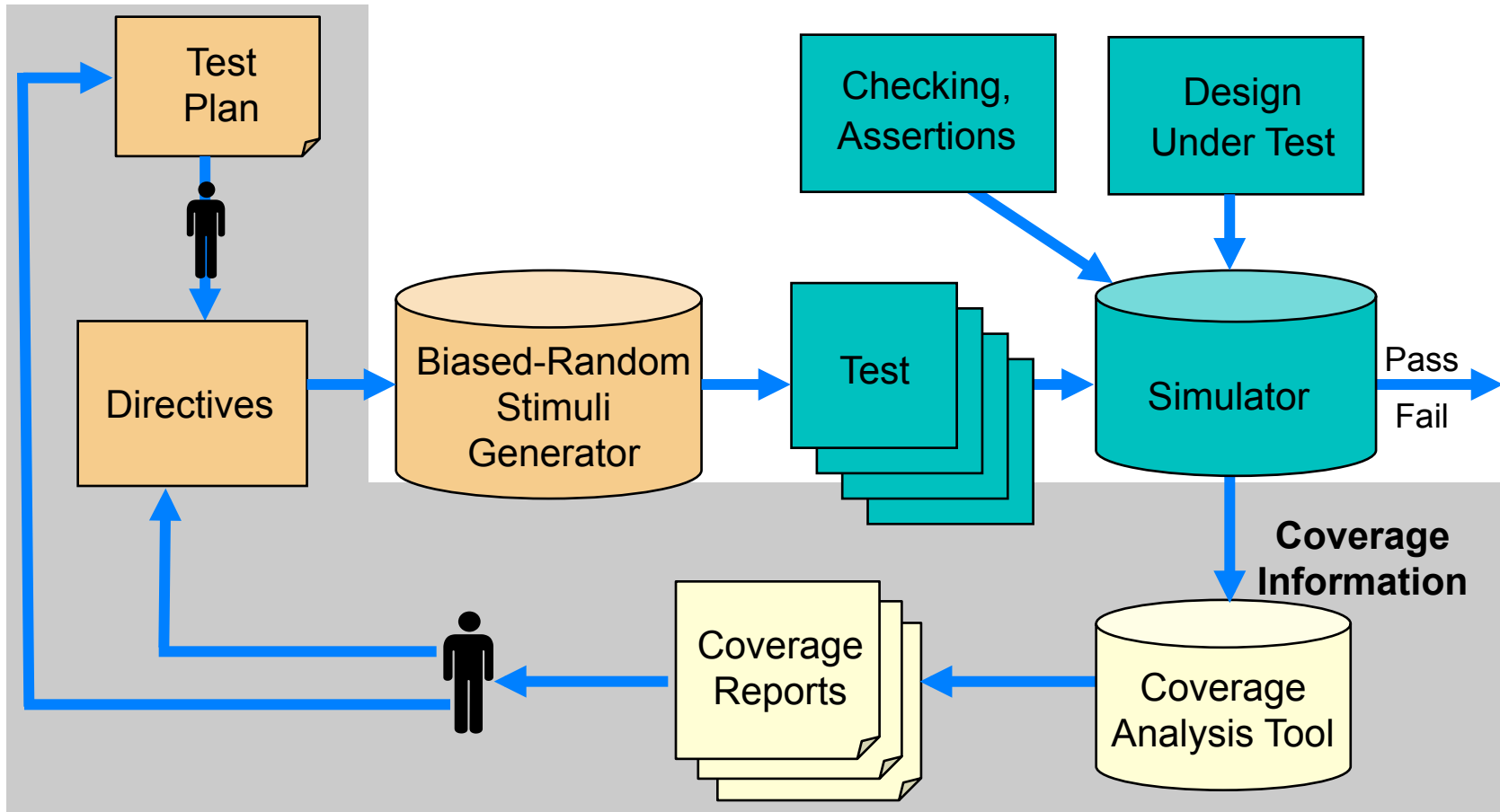
Coverage is part of the Verification Tools.

# Outline

---

- Introduction to coverage
- Code coverage models
- Structural coverage models
- Functional coverage
- Case study and lessons to learn
- Coverage analysis

# Simulation-based Verification Environment



# Why Coverage?

---

- Simulation is based on limited execution samples
  - Cannot run all possible scenarios, but
  - Need to know that all (important) areas of the DUV are verified
- Solution: **Coverage measurement and analysis**
- The main ideas behind coverage
  - Features (of the specification and implementation) are identified
  - Coverage models capture these features

# Coverage Goals

---

- Measure the "quality" of a set of tests
  - NOTE: Coverage gives ability to see what **has not been** verified!
  - Coverage completeness does not imply functional correctness of the design! Why?
- Help create regression suites
  - Ensure that all parts of the DUV are covered by regression suite
- Provide stopping criteria for unit testing
  - Why "only" for unit testing?
- Improve understanding of the design

# Coverage Types

---

- Code coverage
- Structural coverage
- Functional coverage
- Other classifications
  - Implicit vs. explicit
  - Specification vs. implementation

# Code Coverage - Basics

---

- Coverage models are based on the HDL code
  - Implicit, implementation coverage
- Coverage models are **syntactic**
  - Model definition is based on syntax and structure of the HDL
- **Generic models** – fit (almost) any programming language
  - Used in both software and hardware design



# Code Coverage - Scope

---

Code coverage can answer the question:

**“Is there a piece of code that has not been exercised?”**

- Method used in software engineering for some time.
- Have you used gcov?

## **Main problem:**

- **False negative answers can look identical to true negative answers.**

False negative: A bad design is thought to be good.

- **Useful for profiling:**
  - Run coverage on testbench to indicate what areas are executed most often.
  - **Gives insight on what to optimize!**
- Many types of code coverage report metrics/models.

# Types of Code Coverage Models

---

- Control flow
  - Check that the control flow of the program has been fully exercised
- Data flow
  - Models that look at the flow of data in, and between, programs/modules
- Mutation
  - Models that check directly for common bugs by mutating the code and comparing results

# Control Flow Models

---

- Routine (function entry)
  - Each function / procedure is called
- Function call
  - Each function is called from every possible location
- Function return
  - Each return statement is executed
- Statement (block)
  - Each statement in the code is executed
- Branch/Path
  - Each branch in branching statement is taken
    - `if, switch, case, when, ...`
- Expression/Condition
  - Each input in a Boolean expression (condition) takes true and false values
    - (See further details later on MC/DC coverage)
- Loop
  - All possible number of iterations in (Bounded) loops are executed

# Statement/Block Coverage

Measures which lines (statements) have been executed by the verification suite.

```
✓ if (parity==ODD || parity==EVEN) begin
❑ parity_bit = compute_parity(data,parity);
  end
✓ else begin
✓ parity_bit = 1'b0;
  end
✓ #(delay_time);
✓ if (stop_bits==2) begin
✓ end_bits = 2'b11;
✓ #(delay_time);
  end
```

**What do we need to do to get statement coverage to 100%?**

- Why has this never occurred?
- Is it a condition that can never occur? Was it simply forgotten?
- (Dead code can be “ok”!) WHY?

# Path/Branch Coverage

Measures all possible ways to execute a sequence of statements.

- Are all **if/case** branches taken?
- How many execution paths?

```
✓ if (parity==ODD || parity==EVEN) begin
✓ parity_bit = compute_parity(data,parity);
end
✓ else begin
✓ parity_bit = 1'b0;
end
✓ #(delay_time);
✓ if (stop_bits==2) begin
✓ end_bits = 2'b11;
✓ #(delay_time);
end
```

□ □ ✓ ✓

Note: 100%  
statement coverage  
but only 75% path  
coverage!

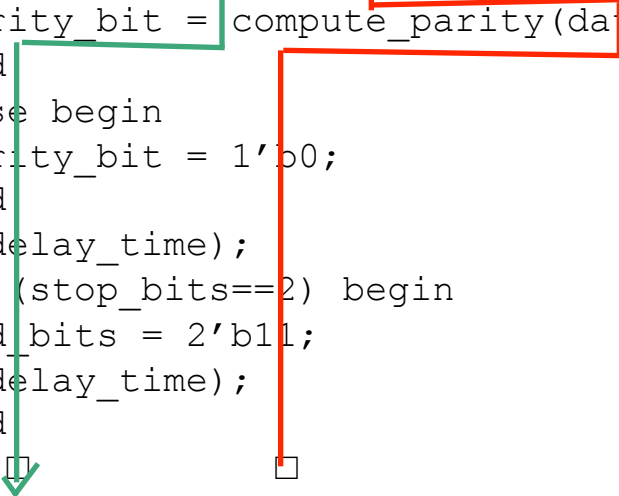
- **Dead code:** default branch on exhaustive **case**
- Don't measure coverage for code that was not meant to run! (tags)

# Expression/Condition Coverage

Measures the various ways Boolean expressions and subexpressions are executed.

- Where a branch condition is made up of a Boolean expression, we want to know which of the inputs have been covered.

```
✓ if (parity==ODD || parity==EVEN) begin
✓ parity_bit = compute_parity(data,parity);
end
✓ else begin
✓ parity_bit = 1'b0;
end
✓ #(delay_time);
✓ if (stop_bits==2) begin
✓ end_bits = 2'b11;
✓ #(delay_time);
end
```

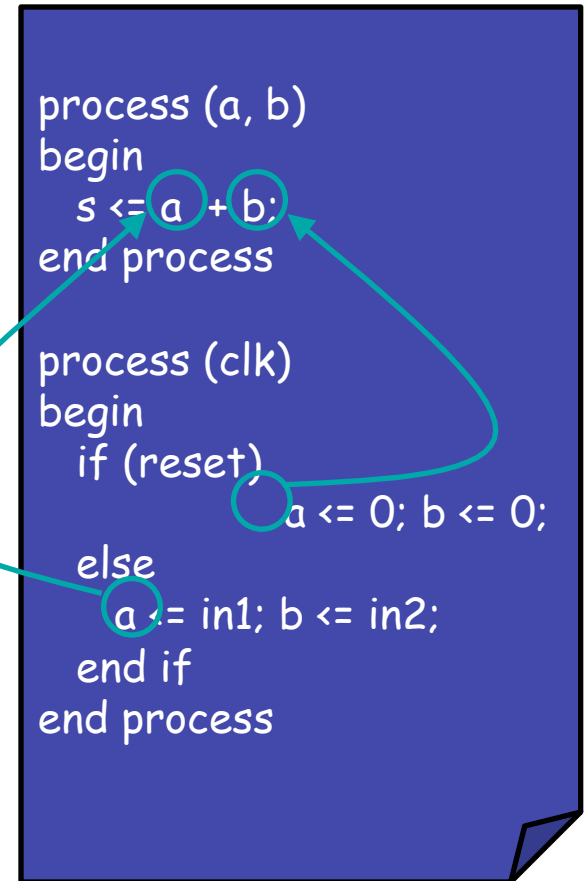


Note: Only 50%  
expression  
coverage!

- **Analysis:** Understand WHY part of an expression was not executed
- **Reaching 100% expression coverage is extremely difficult.**  
(See also MC/DC coverage and use in certification!) ☺

# Data Flow Models

- Coverage models that are based on flow of data during execution
- Each coverage task has two attributes
  - **Define** – where a value is assigned to a variable (signal, register, ...)
  - **Use** – where the value is being used
- Types of dataflow models
  - C-Use – Computational use
  - P-Use – Predicate use
  - All Uses – Both P and C-Uses



# Mutation Coverage

- Mutation coverage is designed to detect simple (typing) mistakes in the code
  - Wrong operator
    - + instead of –
    - >= instead of >
  - Wrong variable
  - Offset in loop boundaries
- A mutation is considered covered if we found a test that can distinguish between the mutation and the original
  - Strong mutation – the difference is visible in the primary outputs
  - Weak mutation – the difference is visible inside the DUV only
- For more on Mutation Coverage see:
  - J Offutt and R.H. Untch. “Mutation 2000: Uniting the Orthogonal”*
- Commercial tools: Certitude by Synopsys
  - <https://www.synopsys.com/verification/simulation/certitude.html>



# Code Coverage Models for Hardware

---

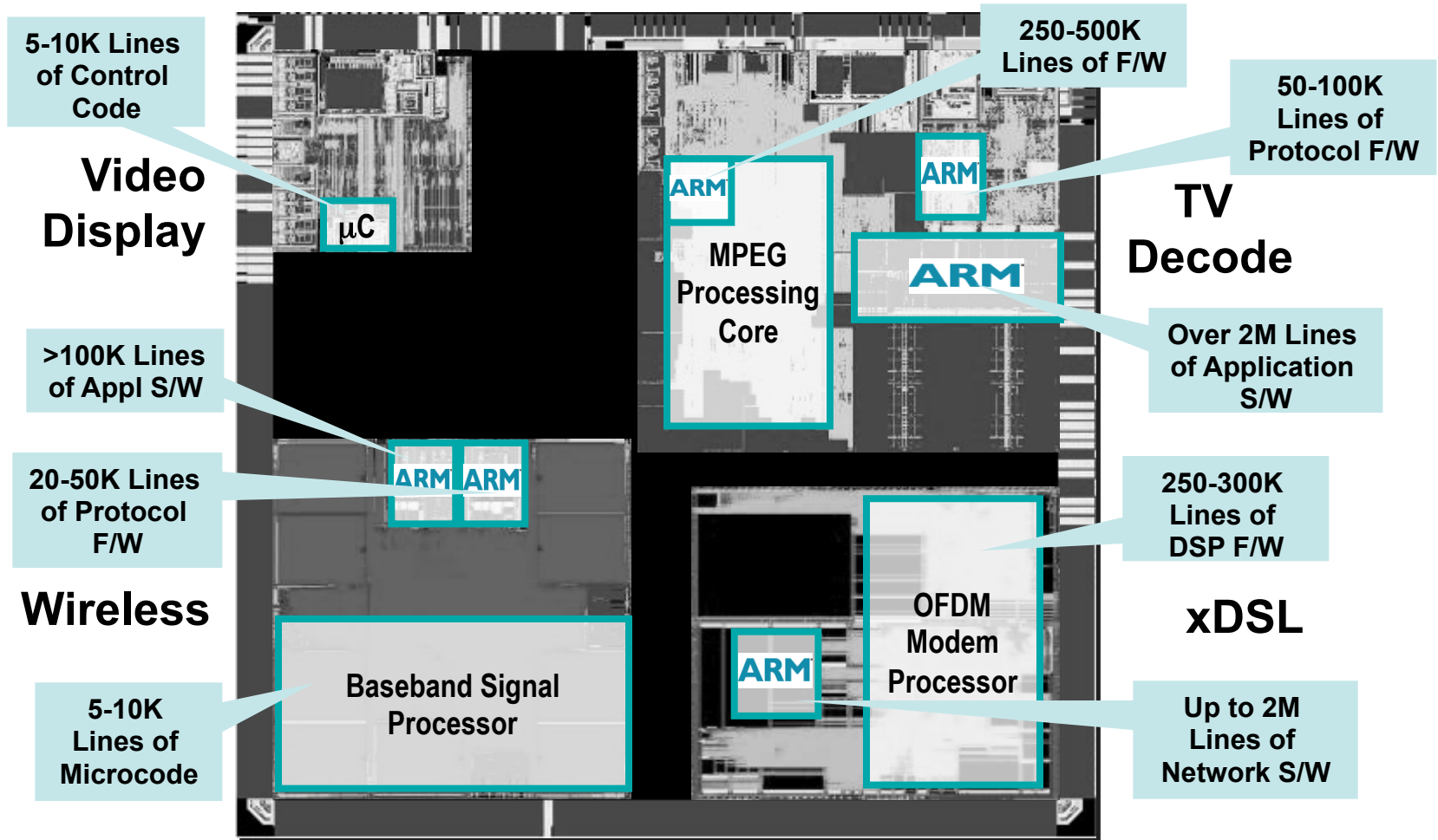
- Toggle coverage
  - Each (bit) signal changed its value from 0 to 1 and from 1 to 0
- All-values coverage
  - Each (multi-bit) signal got all possible values
  - Used only for signals with small number of values
    - For example, state variables of FSMs

# Code Coverage Strategy

- Set **minimum % of code coverage** depending on available verification resources and importance of preventing post tape-out bugs.
  - A failure in low-level code may affect multiple high-level callers.
  - Hence, set a higher level of code coverage for unit testing than for system testing.
- **Generally, 90% or 95% goal for statement, branch or expression coverage.**
  - Some feel that less than 100% does not ensure quality.
  - Beware: Reaching full code coverage closure can cost a lot of effort!
  - This effort could be more wisely invested into other verification techniques.
- **Avoid setting a goal lower than 80%.**

Literature: *[J Barkley. Why Statement Coverage Is Not Enough. A practical strategy for coverage closure., TransEDA.]*

# Increasing Design Complexity



**Multiple Power Domains, Security, Virtualisation**  
**Nearly five million lines of code to enable Media gateway**

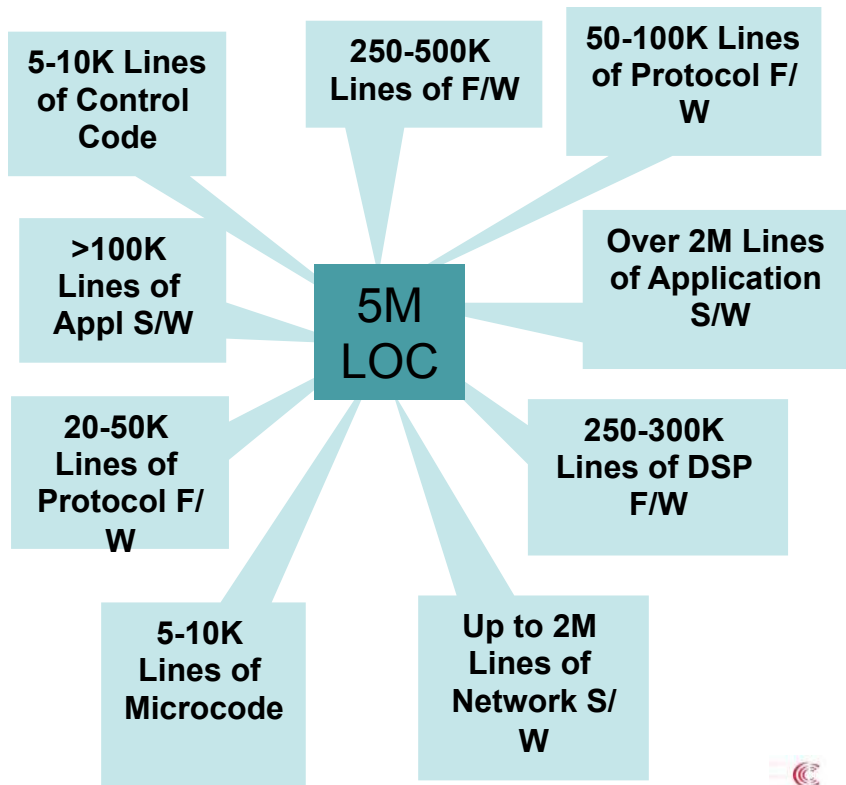
# Increasing Design Complexity

LOC count:

10K  
100K  
50K  
10K  
500K  
100K  
2M  
300K  
2M

TOTAL: ~5M LOC

**At 95% coverage, this  
leaves 250K LOC  
not exercised during  
verification!**



# Modified Condition/Decision (MC/DC) Coverage

Tutorial on MC/DC Coverage: “A Practical Tutorial on Modified Condition/Decision Coverage” by Kelly Heyhurst et. al.

[http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20010057789\\_2001090482.pdf](http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20010057789_2001090482.pdf)

## Terminology:

The literals/inputs in a Boolean expression are termed **conditions**.  
The output of a Boolean expression is termed **decision**.

- **Decision coverage = branch coverage**

- Requires that each decision toggles between true and false.

- e.g. in  $a \ || \ b$  vectors TF and FF satisfy this requirement

- **Condition coverage (also called expression coverage)**

- Requires that each condition (literal in a Boolean expression) takes all possible values at least once, but does not require that the decision takes all possible outcomes at least once.

- e.g. in  $a \ || \ b$  vectors TF and FT satisfy this requirement

# Modified Condition/Decision (MC/DC) Coverage

- **Condition/Decision coverage**
  - Requires that each condition toggles and each decision toggles,
    - e.g. in  $a \ || \ b$  vectors TT and FF satisfy this requirement
- **Multiple Condition / Decision coverage**
  - Requires that all conditions and all decisions take all possible values.
  - This is exhaustive expression coverage.
    - e.g. in  $a \ || \ b$  vectors TT, TF, FT and FF satisfy this requirement
  - **Exponential growth in number of conditions.**

# Modified Condition/Decision (MC/DC) Coverage

- **MC/DC Coverage** requires that each condition be shown to **independently** affect the outcome of the decision while fulfilment of the condition/decision coverage requirements.
  - e.g. in  $a \mid \mid b$  vectors TF, FT and FF satisfy this requirement
- The independence requirement ensures that the effect of each condition is tested relative to the other conditions.
- A minimum of  $(N + 1)$  test cases for a decision with  $N$  inputs is required for MC/DC in general.
- In some tools MC/DC coverage is referred to as **Focused Expression Coverage (fec)**.

# Structural Coverage

---

- Implicit coverage models that are based on **common structures in the code**
  - FSMs, Queues, Pipelines, ...
- The **structures are extracted automatically** from the design and pre-defined coverage models are applied to them
- Users may refine the models
  - Define illegal events

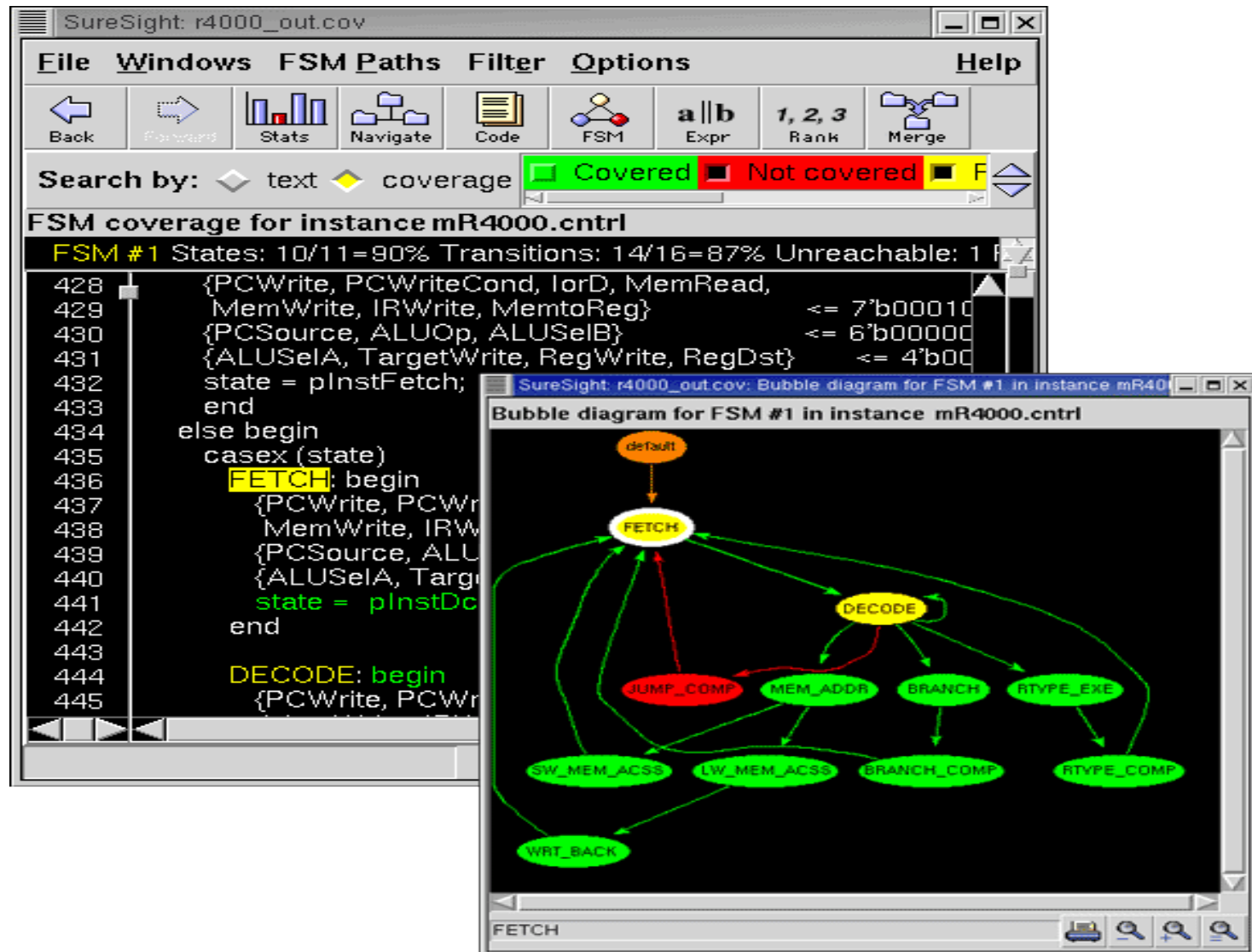


# State-Machine Coverage

---

- State-machines are the essence of RTL design
- FSM coverage models are the most commonly used structural coverage models
- Types of coverage models
  - State
  - Transition (or arc)
  - Path

# FSM Coverage Report



# Code Coverage - Limitations

- Coverage questions not answered by code coverage tools
  - Did every instruction take every exception?
  - Did two instructions access the register at the same time?
  - How many times did cache miss take more than 10 cycles?
  - Does the implementation cover the functionality specified? [Need RBT!]
  - ...(and many more)
- Code coverage indicates how thoroughly the test suite exercises the source code!
  - Can be used to identify outstanding corner cases
- Code coverage lets you know if you are not done!
  - It does not indicate anything about the **functional correctness** of the code!
- **100% code coverage does not mean very much.** ☹
- Need another form of coverage!

# Functional Coverage

---

- It is important to cover the **functionality** of the DUV.
  - Most functional requirements can't easily be mapped into lines of code!
- **Functional coverage models** are designed to assure that various aspects of the functionality of the design are verified properly, they link the requirements/specification with the implementation
- Functional coverage models are specific to a given design or family of designs
- Models cover
  - The inputs and the outputs
  - Internal states or microarchitectural features
  - Scenarios
  - Parallel properties
  - Bug Models

# Functional Coverage Model Types

- **Discrete set of coverage tasks**
  - Set of unrelated or loosely related coverage tasks often derived from the requirements/specification
  - Often used for corner cases
    - Driving data when a FIFO is full
    - Reading from an empty FIFO
  - In many cases, there is a close link between functional coverage tasks and assertions
- **Structured coverage models**
  - The coverage tasks are defined in a structure that defines relations between the coverage tasks
    - Allow definition of similarity and distance between tasks
    - Most commonly used model types
      - Cross-product
      - Trees
      - Hybrid structures

# Cross-Product Coverage Model

---

*[O Lachish, E Marcus, S Ur and A Ziv. Hole Analysis for Functional Coverage Data. In proceedings of the 2002 Design Automation Conference (DAC), June 10-14, 2002, New Orleans, Louisiana, USA.]*

A cross-product coverage model is composed of the following parts:

1. A semantic **description** of the model (story)
2. A list of the **attributes** mentioned in the story
3. A set of all the **possible values** for each attribute (the attribute value **domains**)
4. A list of **restrictions** on the legal combinations in the cross-product of attribute values

# Example: Cross-Product Coverage Model 1

---

## **Design:** switch/cache unit

[G Nativ, S Mittermaier, S Ur and A Ziv. *Cost Evaluation of Coverage Directed Test Generation for the IBM Mainframe. In Proceedings of the 2001 International Test Conference, pages 793-802, October 2001.*]

**Motivation:** Interactions of core processor unit **command-response** sequences can create complex and potentially unexpected conditions causing contention within the **pipes** in the switch/cache unit when many **core processors** (CPs) are active.

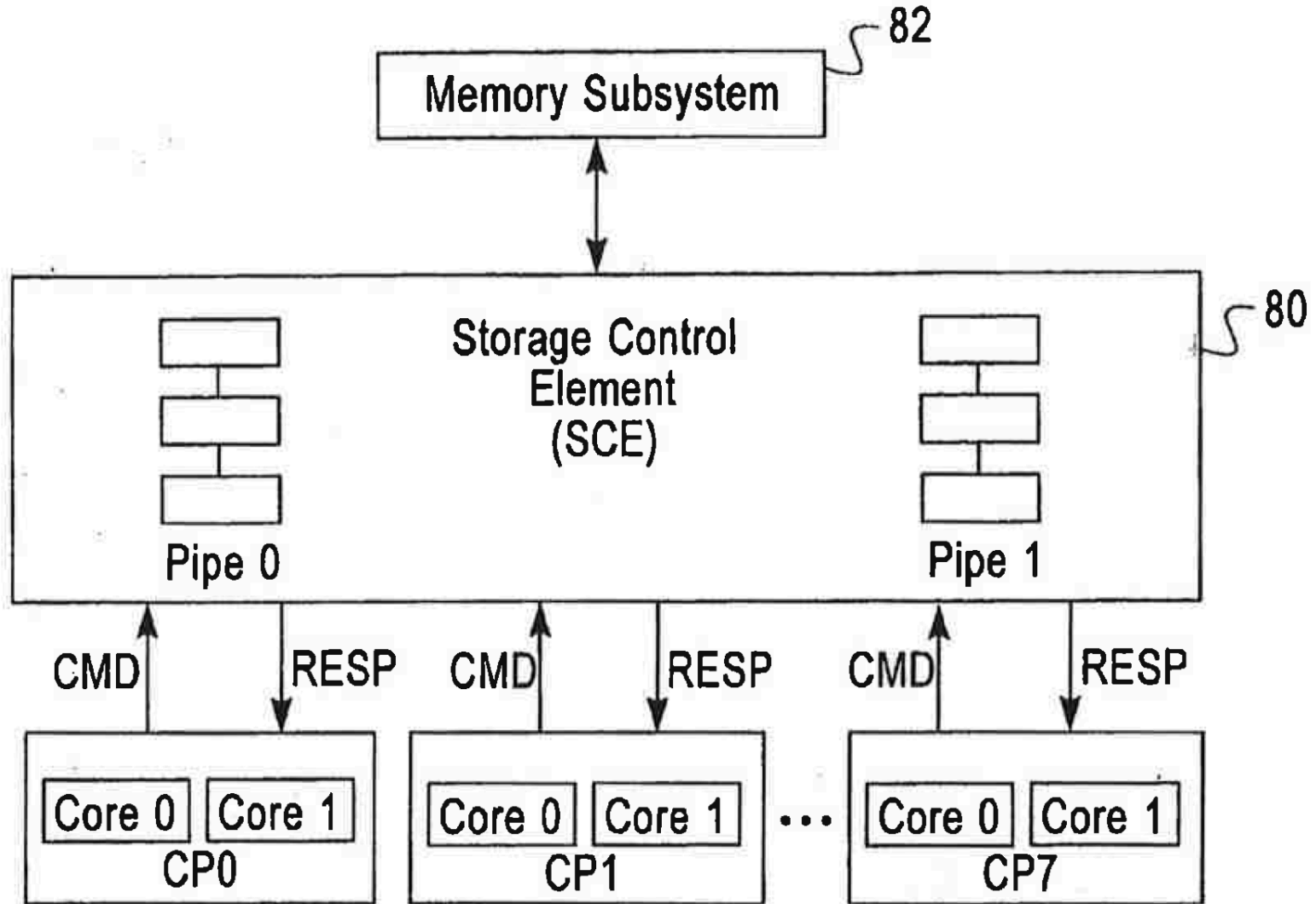
All conditions must be tested to gain confidence in design correctness.

## **Attributes relevant to command-response events:**

- Commands - CPs to switch/cache [31]
- Responses - switch/cache to CPs [16]
- Pipes in each switch/cache [2]
- CPs in the system [8]
- (Command generators per CP chip [2])

How big is the coverage space, i.e. how many coverage tasks?

# Switch/Cache Unit





# Example: Cross-Product Coverage Model 2

## Size of coverage space:

- Coverage space is formed by **cross-product (or, more formally, the Cartesian product) over all attribute value domains.**
- Size of cross-product is product of domain sizes:
  - $31 \times 16 \times 2 \times 8 \times 2 = 15872$
- Hence, there are 15872 coverage tasks.

## Example coverage task:

(Command=20, Response=01, Pipe=1, CP=5, CG=0)

## Are all of these tasks reachable/legal?

- Restrictions on the coverage model are:
  - possible responses for each command
  - unimplemented command/response combinations
  - some commands are only executed in pipe 1
- After applying restrictions, there are 1968 legal coverage tasks left.
- **Make sure you identify & apply restrictions before you start!**

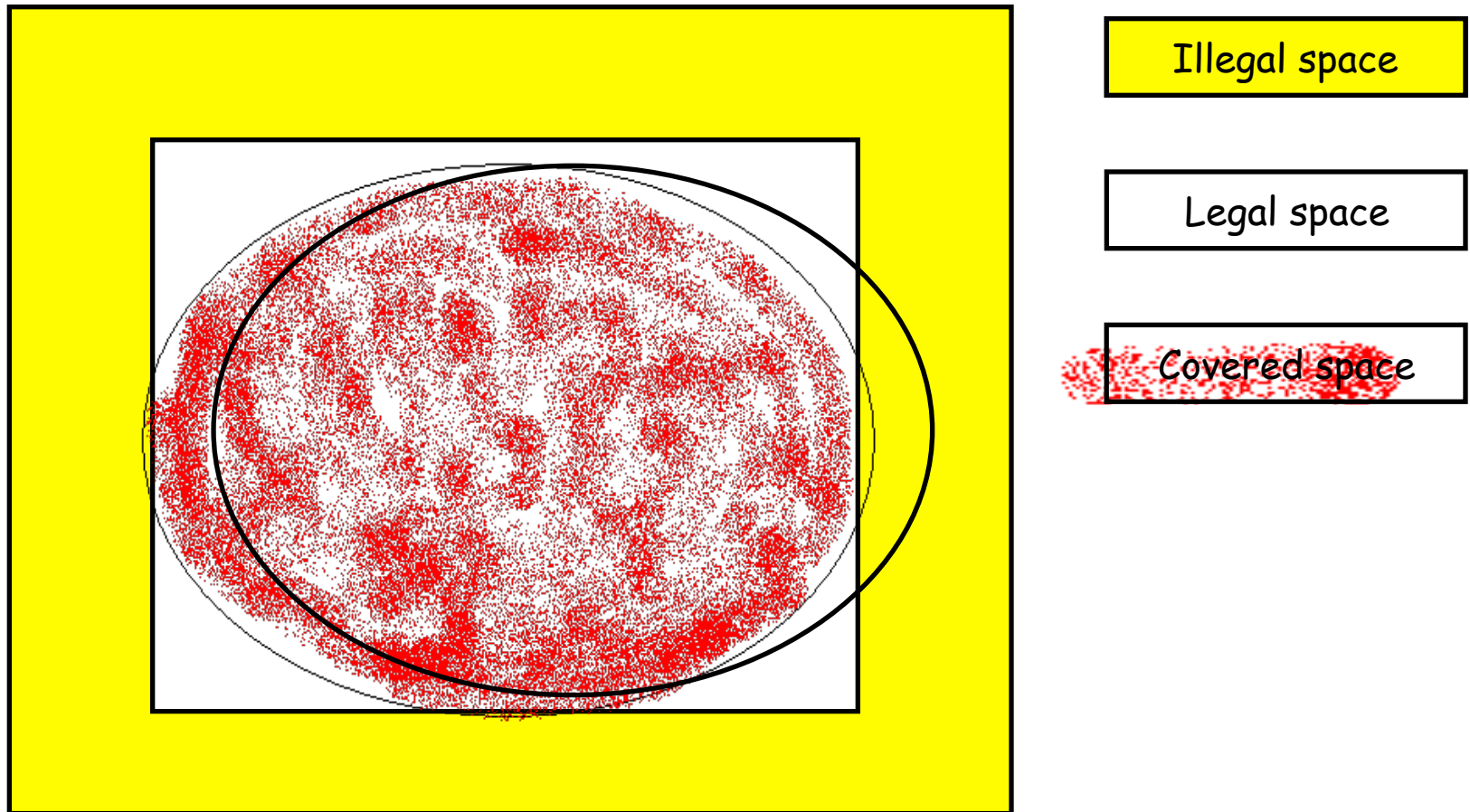
# Defining the Legal and Interesting Spaces

---

## In Practice:

- Boundaries between legal and illegal coverage spaces are often not well understood
- The design and verification team create initial spaces based on their understanding of the design
- Coverage feedback modifies the space definition
- **Sub-models** are used to economically check and refine the spaces
  - Easy to define as these are sub-crosses!
- Interesting spaces tend to **change often** due to shift in focus in the verification process

# Legal Spaces Are Self-correcting



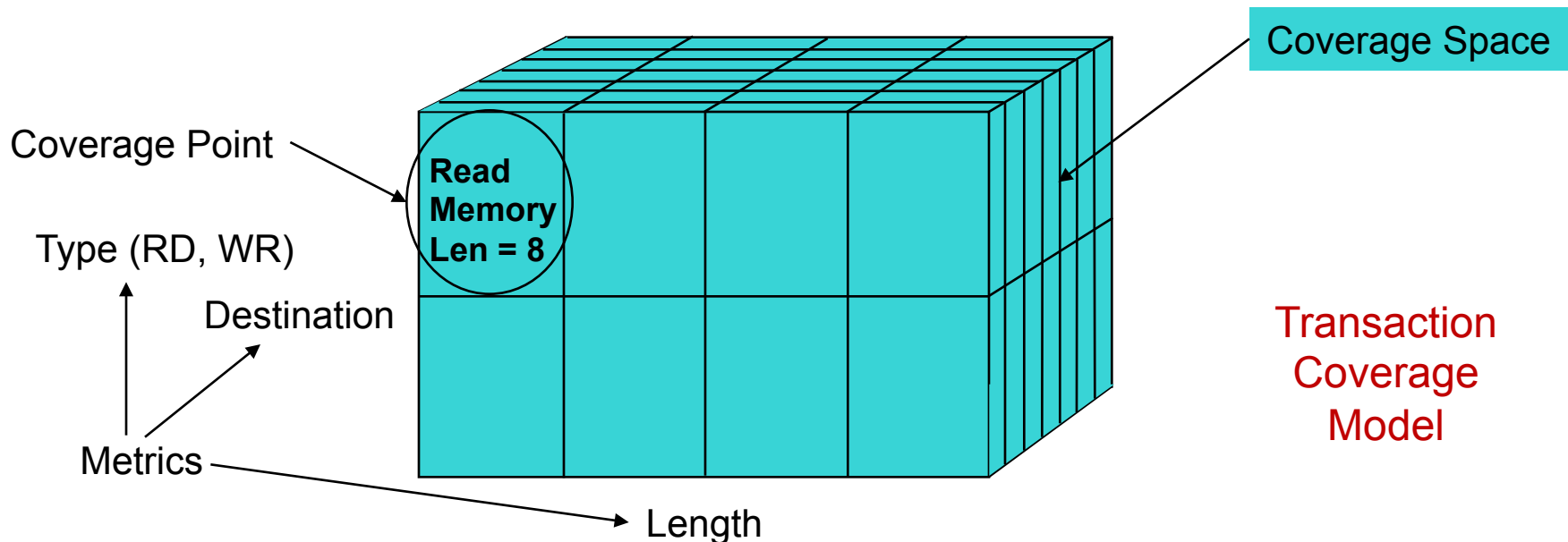
# Cross-Product Coverage more formally

- Functional cross-product coverage models can be defined using **multi-dimensional coverage spaces**.
- A **functional coverage space**  $C_m$  is defined as the Cartesian product over  $m$  signal domains  $D_0; \dots; D_{m-1}$ .
  - $C_m = D_0 \times \dots \times D_{m-1}$
- Let  $||D_k|| = d_k$  denote the **size of domain**  $D_k$ .
- The functional coverage space  $C_m$  contains  $||C_m|| = ||D_0|| * \dots * ||D_{m-1}|| = d$  distinct **coverage points**  $p_0; \dots; p_{d-1}$ .
- A **coverage point**  $p_i$  with  $i \in \{0; \dots; d-1\}$  is characterized by an  **$m$ -tuple of values**  $p_i = (v_0; \dots; v_{m-1})$ , where  $p_i[k] = v_k$  and each  $v_k \in D_k$ , for  $k \in \{0; \dots; m-1\}$ .

**Formalization facilitates automation of coverage analysis e.g. identification of coverage gaps.**

# Coverage Terminology

- **coverage model** *n. 1. A set of legal and interesting coverage points in the coverage space.*
- **coverage point** *n. 1. A point within a multi-dimensional coverage space. 2. An event of interest that can be observed during simulation.*



# Cross-Product Models In e

## Verification Languages such as e support cross-product coverage models:

- The **story** is hidden in the **event**
- The **attributes** and their **values** are defined in the **coverage items**
- **Legal and interesting space** are defined using the **illegal** and **ignore** constructs
  - Restrictions can be defined on the coverage items and the cross itself

```
struct instruction {  
    opcode: [NOP, ADD, SUB,  
            AND, XOR];  
    operand1 : byte;  
  
    event stimulus;  
  
    cover stimulus is {  
        item opcode;  
        item operand1;  
        cross opcode, operand1  
            using ignore = (opcode == NOP);  
    };  
};
```

# New: Situation Coverage

	T	┐	└	┌	—	↑	↓	+	⊥	└	┌	┐	└	┌	└
Car															
Bike															
HGV															
Ped															

*Alexander, Rob; Hawkins, Heather Rebecca; Rae, Andrew John*  
***Situation coverage – a coverage criterion for testing autonomous robots.***  
*Department of Computer Science, University of York, 2015. 21 p.*

# Summary: Functional Coverage

---

Determines whether the **functionality** of the DUV was verified.

- Functional coverage models are **user-defined**.
  - (specification driven)
  - This is a skill. It needs (lots of) experience!
  - Focus on **control signals**. WHY?
- **Strengths:**
  - High expressiveness: cross-correlation and multi-cycle scenarios.
  - Objective measure of progress against verification plan.
  - Can identify coverage holes by crossing existing items.
  - Results are easy to interpret.
- **Weaknesses:**
  - Only as good as the coverage metrics.
  - To implement the metrics, engineering effort is required and a lot of expertise.



# Summary: Code Coverage

---

Determines if all the **implementation** was verified.

- Models are implicitly defined by the source code.
  - (implementation driven)
  - statement, path, expression, toggle, etc.
- **Strengths:**
  - Reveals unexercised parts of design.
  - May reveal gaps in functional verification plan.
  - No manual effort is required to implement the metrics. (Comes for free!)
- **Weaknesses:**
  - No cross correlations.
  - Can't see multi-cycle/concurrent scenarios.
  - Manual effort required to interpret results.

# Summary: Coverage Models

- Do we need both code and functional coverage? YES!

Functional Coverage	Code Coverage	Interpretation
Low	Low	There is verification work to do.
Low	High	Multi-cycle scenarios, corner cases, cross-correlations still to be covered.
High	Low	Verification plan and/or functional coverage metrics inadequate. Check for “dead” code.
High	High	High confidence in quality.

- Coverage models complement each other!
- No single coverage model is complete on its own.

# Case Studies

# The Coverage Process in Practice

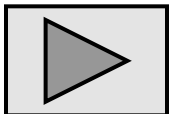
---

## Examples:

- Verifying interdependency in a PowerPC processor
- Pipeline of Branch unit in S/390 system

(Thanks to Avi Ziv from IBM Research Labs in Haifa for sharing these.)

Coverage Analysis

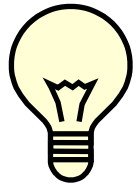


Coverage Closure  
(now part of the  
“Closing the Cycle” lecture)

# Example 1: Interdependency in a PowerPC Processor

---

- Interdependencies between instructions in the pipeline of a processor create interesting testing scenarios

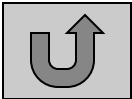


- They activate many **microarchitectural mechanisms**, such as forwarding and stalling
- Studies have shown that they are the **source of many bugs** in processor designs
- Functionality at this level is often related to increasing **processor performance**

# Lesson No. 1

---

- *Define coverage models in interesting areas in the design*
  - Bug prone, New logic, Complex algorithm
- In our case:
  - Register interdependency activates many pipeline mechanisms, such as forwarding and stalling
  - Coverage model aims to ensure that all forward and stall mechanisms are activated



# First Approach – Black Box Model

---

- The motivation (story):

Verify all dependency types of a resource (register) relating to all instructions

- The **semantics** of the coverage tasks:

A coverage task is a quadruplet  $(I_i, I_k, R, DT)$ , where Instruction  $I_i$  is followed by Instruction  $I_k$ , and both share Resource  $R$  with Dependency Type  $DT$

- The attributes:
  - $I_i, I_k$  - Instruction: add, sub, ...
  - $R$  - Register (resource): G1, G2, ...
  - $DT$  - Dependency Type:
    - WW, WR, RW, RR and ???

# First Approach – Black Box Model

---

- The motivation (story):

Verify all dependency types of a resource (register) relating to all instructions

- The **semantics** of the coverage tasks:

A coverage task is a quadruplet  $(I_i, I_k, R, DT)$ , where Instruction  $I_i$  is followed by Instruction  $I_k$ , and both share Resource  $R$  with Dependency Type  $DT$

- The attributes:

- $I_i, I_k$  - Instruction: add, sub, ...
- $R$  - Register (resource): G1, G2, ...
- $DT$  - Dependency Type:
  - WW, WR, RW, RR and None



# More Semantics

---

- The semantics provided so far is too coarse
  - What if  $I_i$  is the first instruction in the test and  $I_k$  is the 1000 instruction?
- Need to refine the semantics to improve probability of hitting interesting events
- Additional semantics
  - The distance between the instructions is no more than 5
  - The first instruction is at least the 6th

# The Legal Space

---

- Not all combinations are valid
  - Not all instructions read from registers
  - Not all instructions write to registers
  - Fixed point instructions cannot share FP (floating point) registers
  - ... and more

# Space and Model Size

- PowerPC has

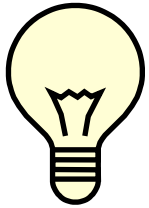
- ~400 instructions

- (actually this is an old number, current PowerPC has close to 1000 instructions)

- ~100 registers

- Coverage space size is  $400 \times 400 \times 100 \times 5 = 80,000,000$  tasks

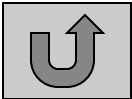
- Even after all restrictions are applied, the model size is still 200,000 tasks



# Lesson No. 2

---

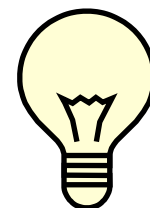
- *Define a model of realistic size*
  - Ensure good coverage can be achieved with simulation resources
  - Group similar cases together to reduce model size
- In our case:
  - Original space size is  
 $(400 \times 400 \times 100 \times 5) = 80,000,000$  tasks
  - Many instructions behave similarly in the pipe
    - For example add and sub
  - Many registers are activated in the same way
    - All general purpose registers, all floating-point registers
  - Grouping similar instructions together helps to reduce the model size to a manageable size



# Coverage Results

---

- A random test generator was used to generate tests that achieved 100% coverage
- Testing the generated tests against the forwarding and stalling mechanisms of a specific processor showed that many such mechanisms were not activated by the tests



# Lesson No. 3

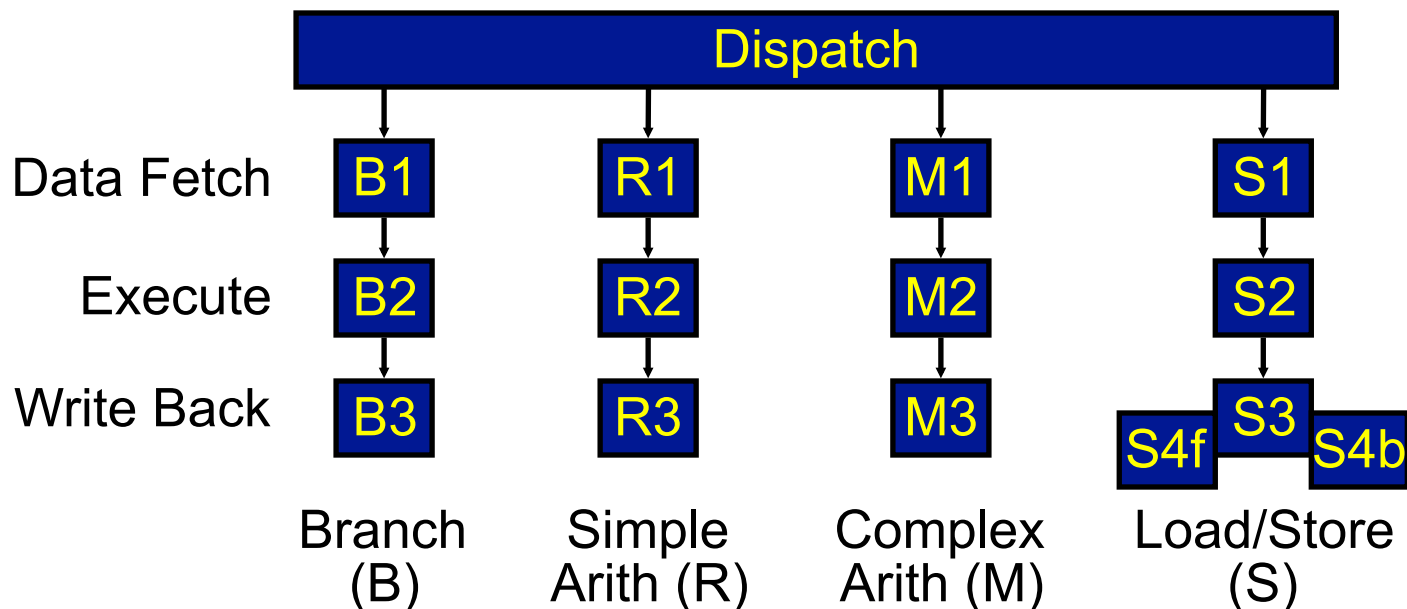
---

- *Define coverage models at the proper level of abstraction for the coverage tasks*
- In our case:
  - Forwarding and stalling are **microarchitectural** mechanisms, so the coverage model should be defined at the microarchitectural level
- In general:
  - Microarchitecture is the place to look for coverage models
    - This is where the complexity of the design hides
      - Architecture is not detailed enough
      - Implementation is too messy



# Grey Box Model

- Microarchitectural model for a specific Processor
  - Multithreaded
  - In-order execution
  - Up to four instructions dispatched per cycle



# Model Details

---

- Model contains 7 attributes
  - Type, pipe and stage of first instruction (I1 ,P1 ,S1)
  - Same attributes for second instruction (I2, P2, S2)
  - Type of dependency between the instructions
    - RR, RW, WR, WW, None
- Grouping is done in a similar way to the architectural model
- Many restrictions exist
  - I1 is simple fixed point  $\rightarrow$  P1 is R or M
  - P1 is not S  $\rightarrow$  S1 is 1, 2, or 3
- After restrictions, 4418 tasks are legal



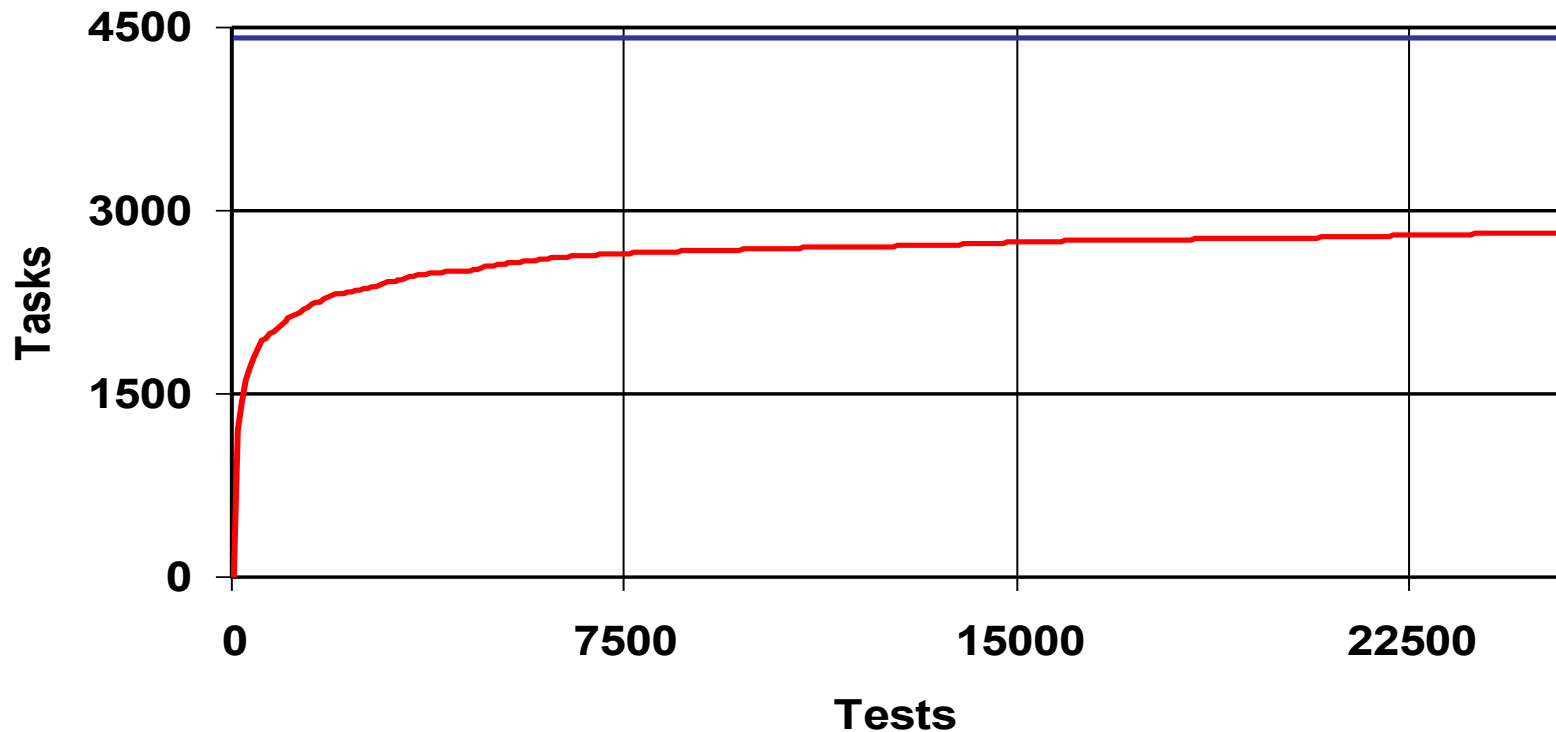
# Coverage Measurement

---

- Make sure that you measure what you really want and what really happens
- Use simpler environment and models to **test and debug the measurement system**
  - Hierarchy of models
    - All instructions
    - All pipe stages
  - Controlled simulation

# Analysis of Interdependency Model

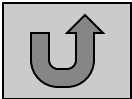
- After 25,000 tests 2810 / 4418 tasks were covered (64%)



# Lesson No. 4

---

- *Coverage analysis is more than a single number*
- In our case:
  - 64% is not bad but
  - Progress report shows that coverage is progressing slowly
  - Hole analysis finds big areas that are covered very lightly
  - Analysis found some problems in test generators



# Analysis of Interdependency Model

---

- Hole analysis detected two major areas that are lightly covered



- Stages S4f and S4b that are specific to thread switching are almost always empty
  - Reason: not enough thread switches during tests
- The address-base register in the store-and-update instruction is not shared with other registers in the test
  - Reason: bug in the test generator that didn't consider the register as a modified register

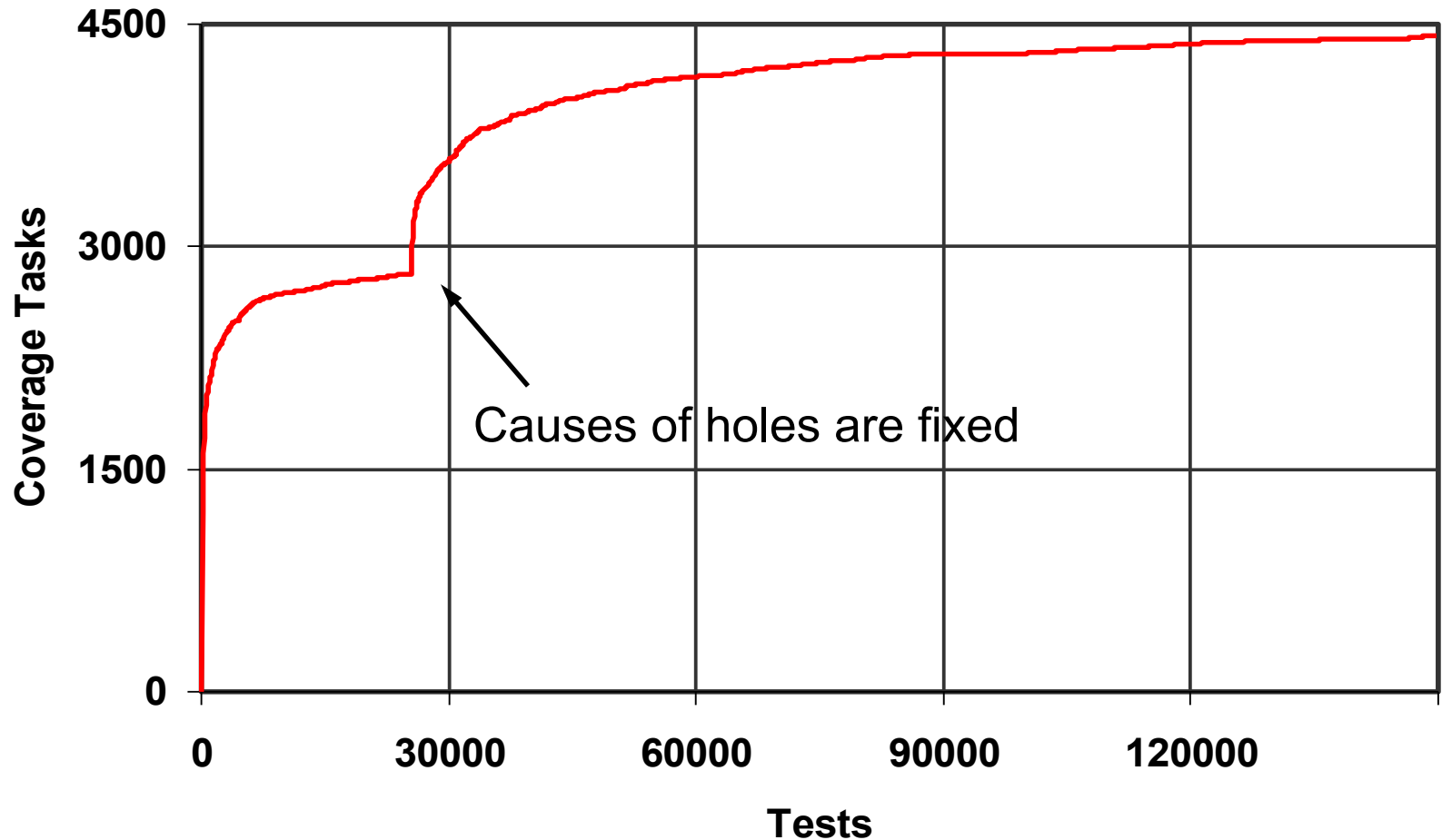
# Lesson No. 5

---

- *Look for large uncovered areas*
  - Can indicate problems in the testing
  - Or missing restrictions
- Constantly update the coverage models
  - Makes coverage picture clearer
- In our case:
  - Two large holes caused by problems in the test generator and test specification



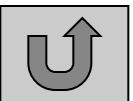
# Coverage Progress



# Architecture vs. Microarchitecture

---

- Architecture
  - No implementation details
  - Easy to share between designs
  - Temporal model
- Microarchitecture
  - Pipe implementation knowledge is needed
  - Access to microarchitectural mechanisms is needed
    - White box or at least grey box
    - More for observability than for controllability (Why?)
  - Snapshot model



# Example 2: S/390 Branch Unit

---

- Unit handles branch prediction and execution of branch instructions
- Contains
  - Nine stage complex pipe
    - More than one instruction at the same time in some stages
    - Instructions can enter the pipe at two places
  - Branch history tables
  - and more
- 2 PY spent on verification
- Done by experts with experience with similar designs
- About 100,000 tests per day



# Coverage Models for Branch Unit

- Several models defined



- Access to branch tables
- Flow of a branch in the pipe
- State of the pipe

- State of the pipe model



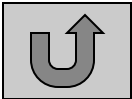
- Attributes contain
  - Location and type of each branch in the pipe in a given cycle
  - Reset signal
- Model size:
  - Without restrictions ~ 15,000,000
  - With restrictions ~ 1400



# Lesson No. 6

---

- *Define families of coverage models that represent different views of the design*
  - Help capture all the functionality with a small number of coverage tasks
  - Analysis of one model can help understanding behavior of another
- In our case:
  - Two views of pipe functionality
  - Model for the flow of a single instruction in the pipe
  - Model for all instructions in the pipe at a given time



# Lesson No. 7

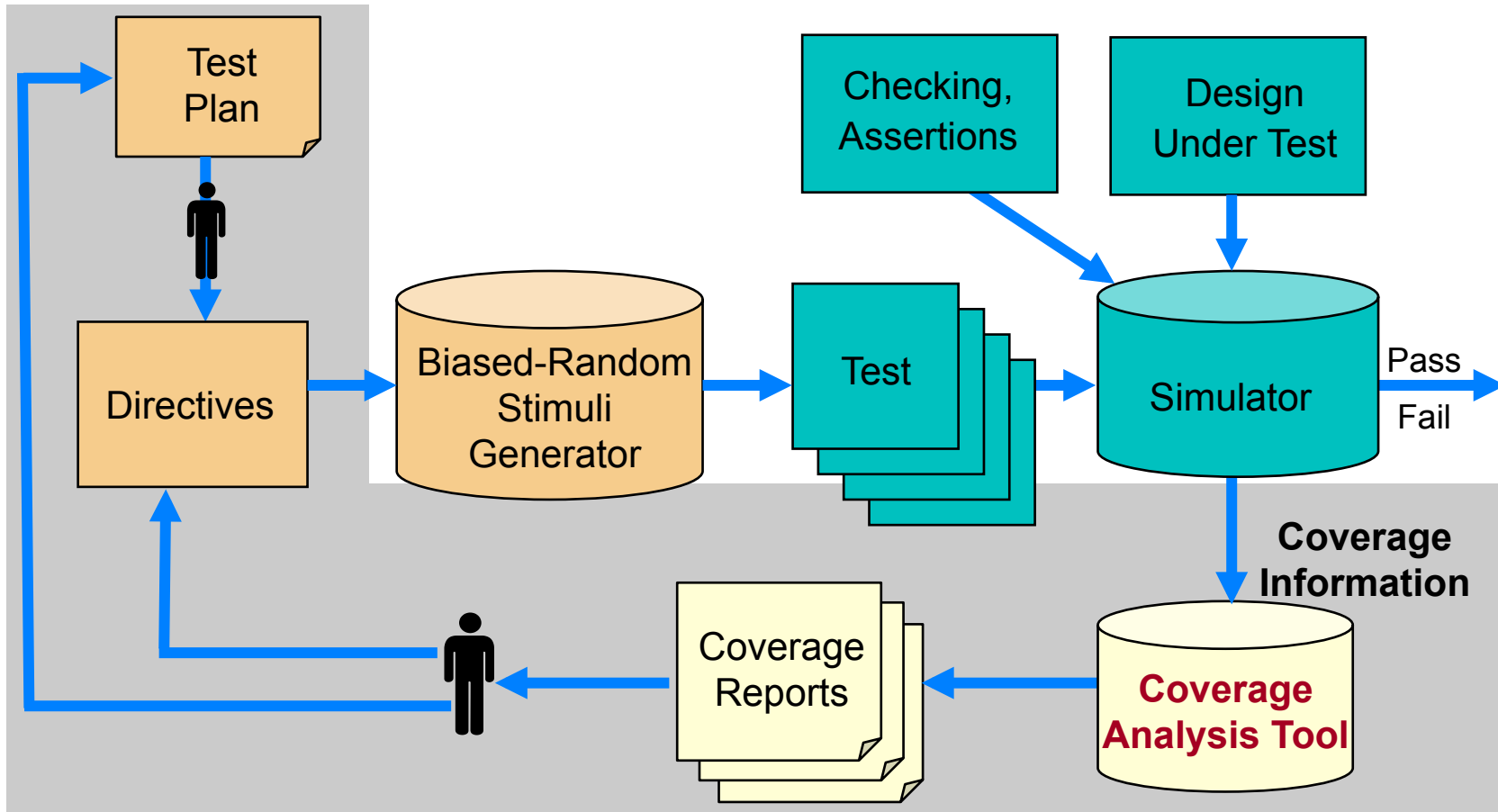
---

- *Look for models that have a view different from the view of the designer*
  - Model definition can lead to better understanding of the design
  - Coverage can lead to unexpected scenarios
- In our case:
  - Designer's view is the flow of instructions in the pipe
  - Model for global pipe state led to accurate analysis of number of instructions in the pipe



# Coverage Analysis

# Coverage Analysis



# Why Coverage Analysis

---

- The main goals of the coverage process are
  - Monitor the quality of the verification process
  - Identify unverified and lightly verified areas
  - Help understanding of the verification process
- **Coverage analysis** helps closing the loop from coverage measurement to the verification plan and test generation

# Coverage Analysis Goals

---

- **Conflicting goals for coverage analysis:**
  - Want to collect as much data as possible
    - Not to miss important events
  - User needs concise and informative reports
    - Not to get drawn into too much detail
- Different types of users require different types of information
- **Goal:** provide concise and informative reports that address the specific needs of the report user

# Types of Coverage Reports

---

- Status reports
  - Coverage status summary
  - Detailed status reports of covered and uncovered tasks
    - Reports can be adapted to specific user needs
    - Allow interactive navigation between reports to explore coverage state
- Progress reports
  - Progress of coverage over time



# Coverage Status Summary

---

- Provides a short summary of the coverage state
- Provides the overall state of the coverage model (or models)
- Useful for
  - Status meetings and status reports
  - A quick glance at the coverage state

Size of coverage space:	1539648
Number of tasks:	4200
Number of tasks covered:	1273
Percent tasks covered:	30.39524
Number of holes:	2927
Number of illegal tasks:	9
Number of traces measured:	16254
Number of cycles measured:	94231273

# Detailed Status Report

- Provides details on each task in the coverage model
  - Covered or not
  - How many times covered
  - In how many tests covered
  - First and last time covered
  - Coverage goals

...

	Inst1	Inst2	Reg	Dep	goal	Tests covered	Times covered
	Add	Mul	GPR	RR	3	1	2
	Add	Stw	G0	RW	3	13	21
	Add	Mul	GPR	RR	3	1	2
	Add	Stw	G0	RW	3	13	21
	Sub.	Add.	CR	WR	3	2	3
	Mul	Div	GPR	WW	3	0	0
	Ldw	And	GPR	None	3	3	9
	Add	Mul	GPR	RR	3	1	2
	Add	Stw	G0	RW	3	13	21
	Sub.	Add.	CR	WR	3	2	3
	Mul	Div	GPR	WW	3	0	0
	Ldw	And	GPR	None	3	3	9
	FPdiv	FPsub	FPR	WW	3	1	1
	Br	Sub.	CR	RR	3	12	11
	FPdiv	FPsub	FPR	WW	3	1	1
	Br	Sub.	CR	RR	3	12	11
	Sub.	Add.	CR	WR	3	2	3
	Mul	Div	GPR	WW	3	0	0
	Add	Mul	GPR	RR	3	1	2
	Add	Stw	G0	RW	3	13	21
	Sub.	Add.	CR	WR	3	2	3
	Mul	Div	GPR	WW	3	0	0
	Ldw	And	GPR	None	3	3	9
	FPdiv	FPsub	FPR	WW	3	1	1
	Br	Sub.	CR	RR	3	12	11
	Ldw	And	GPR	None	3	3	9
	FPdiv	FPsub	FPR	WW	3	1	1
	Add	Mul	GPR	RR	3	1	2
	Add	Stw	G0	RW	3	13	21
	Sub.	Add.	CR	WR	3	2	3
	Mul	Div	GPR	WW	3	0	0
	Ldw	And	GPR	None	3	3	9
	FPdiv	FPsub	FPR	WW	3	1	1
	Br	Sub.	CR	RR	3	12	11
	Br	Sub.	CR	RR	3	12	11

# Detailed Status Reports

---

- Detailed status reports can provide too much detail even for a moderate coverage model
  - Hard to focus on the areas in the coverage model we are currently interested in
  - Hard to understand the meaning of the coverage information
    - Are we missing something important?
- Solution: **Views into the coverage data**
  - Allow the user to focus on the current area of interest and look at the coverage data with the appropriate level of detail
  - Dynamically define the coverage model

# Types of Coverage Views

---

- Views based on coverage data
  - Counts
  - Dates
- Views based on coverage definition
  - Projection
  - Selection
  - Partitioning
- Other filtering mechanisms

All the above options can be combined

# Projection

- Project the  $n$  dimensional coverage space onto an  $m$  ( $< n$ ) subspace
- Allow users to concentrate on a specific set of attributes
- Help in understanding some of things leading up to the big picture

Instruction	Count	Density
fadd	12321	127/136
fsub	10923	122/136
fmul	4232	94/136
fsqrt	13288	40/56
fabs	9835	38/40

# Selection

- Selects a subset of the values of an attribute
- Allows the report to concentrate on a specific area in the coverage model
- Clears the report from data that is not of interest at the time

Instruction	Count	Density
fmadd	9725	107/136
fmsub	9328	111/136
frsq rte	9792	23/36
fsqrt	13288	40/56

# Partitioning

- Provides a more **coarse-grained view** of the coverage data
- Partitions values of given attributes into non-overlapping sets
  - Example: **Instruction types** -> Arith, Branch, Load, Store, etc

4/12	9/12	9/12
5/12	10/12	8/12
7/12	3/12	9/12
8/12	7/12	10/12

# Automatic Coverage Analysis

---

- Detailed status reports do not always reveal interesting information hidden in the coverage data
  - You need to know where to look
  - You need to know which questions to ask the coverage tool
- Specifically, it is **hard to find large areas of uncovered tasks** in the coverage model



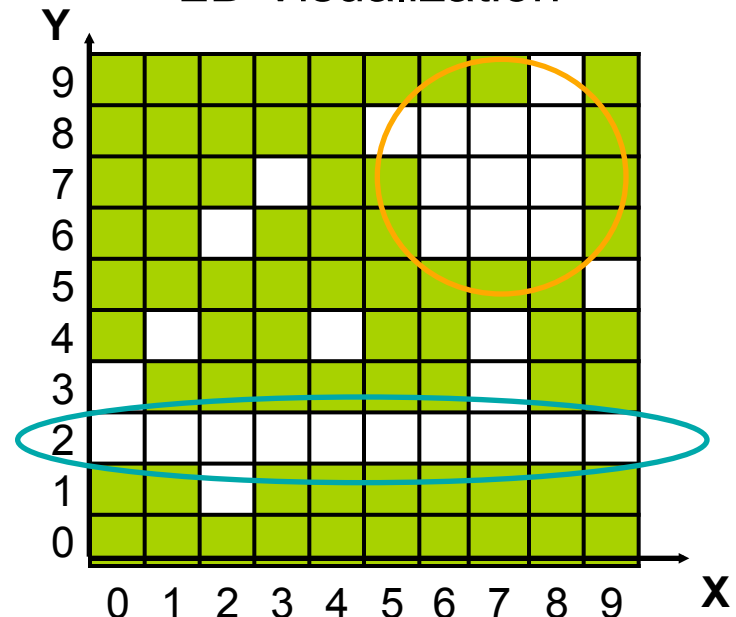
# Large Holes Example

- All combinations of two attributes, X and Y
  - Possible values 0 – 9 for both (100 coverage tasks)
- After a period of testing, 70% coverage is achieved

Uncovered Tasks

●	<table><tr><th>X</th><th>Y</th></tr><tr><td>0</td><td>2</td></tr><tr><td>0</td><td>3</td></tr><tr><td>1</td><td>2</td></tr><tr><td>1</td><td>4</td></tr><tr><td>2</td><td>1</td></tr><tr><td>2</td><td>2</td></tr><tr><td>2</td><td>6</td></tr><tr><td>3</td><td>2</td></tr><tr><td>3</td><td>7</td></tr><tr><td>4</td><td>2</td></tr></table>	X	Y	0	2	0	3	1	2	1	4	2	1	2	2	2	6	3	2	3	7	4	2
X	Y																						
0	2																						
0	3																						
1	2																						
1	4																						
2	1																						
2	2																						
2	6																						
3	2																						
3	7																						
4	2																						
●	<table><tr><th>X</th><th>Y</th></tr><tr><td>4</td><td>4</td></tr><tr><td>5</td><td>2</td></tr><tr><td>5</td><td>8</td></tr><tr><td>6</td><td>2</td></tr><tr><td>6</td><td>6</td></tr><tr><td>6</td><td>7</td></tr><tr><td>6</td><td>8</td></tr><tr><td>7</td><td>2</td></tr><tr><td>7</td><td>3</td></tr><tr><td>7</td><td>4</td></tr></table>	X	Y	4	4	5	2	5	8	6	2	6	6	6	7	6	8	7	2	7	3	7	4
X	Y																						
4	4																						
5	2																						
5	8																						
6	2																						
6	6																						
6	7																						
6	8																						
7	2																						
7	3																						
7	4																						
●	<table><tr><th>X</th><th>Y</th></tr><tr><td>7</td><td>6</td></tr><tr><td>7</td><td>7</td></tr><tr><td>7</td><td>8</td></tr><tr><td>8</td><td>2</td></tr><tr><td>8</td><td>6</td></tr><tr><td>8</td><td>7</td></tr><tr><td>8</td><td>8</td></tr><tr><td>8</td><td>9</td></tr><tr><td>9</td><td>2</td></tr><tr><td>9</td><td>9</td></tr></table>	X	Y	7	6	7	7	7	8	8	2	8	6	8	7	8	8	8	9	9	2	9	9
X	Y																						
7	6																						
7	7																						
7	8																						
8	2																						
8	6																						
8	7																						
8	8																						
8	9																						
9	2																						
9	9																						

2D Visualization



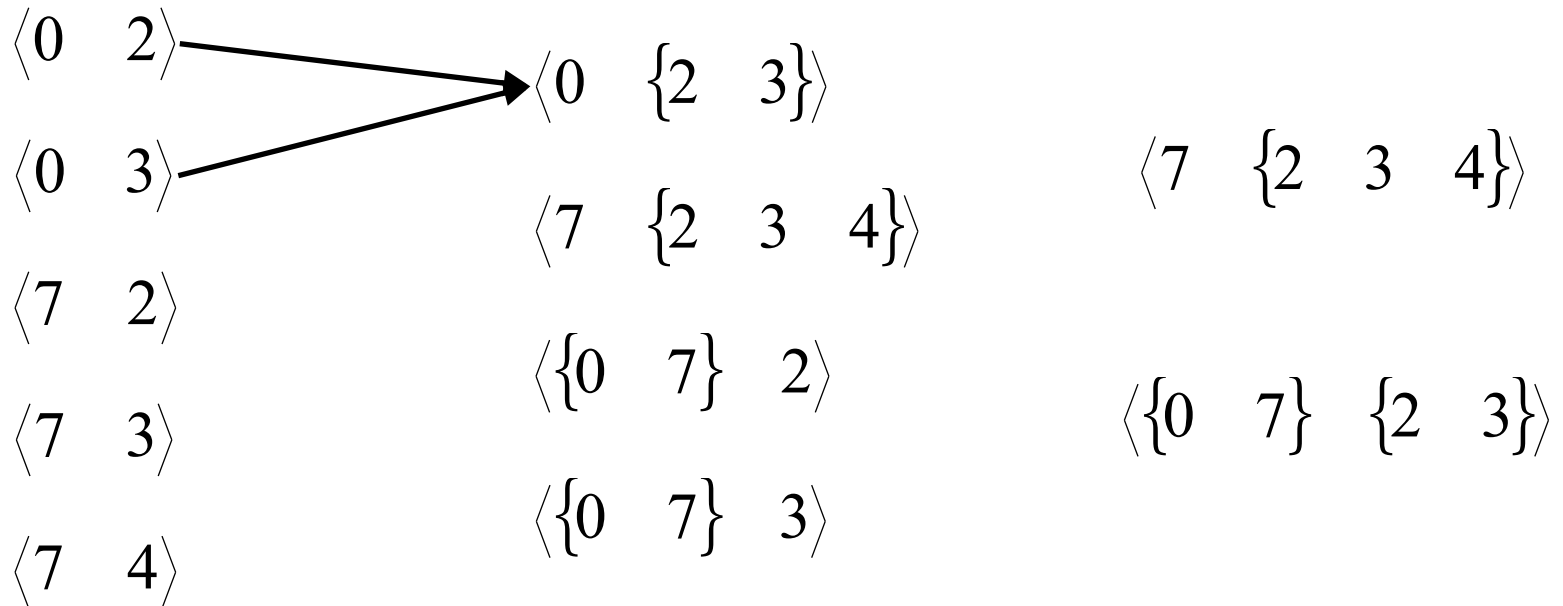
# Hole Analysis Algorithms

---

- Try to find large areas in the coverage space that are not covered
- Use basic techniques to combine sets of uncovered events into large meaningful holes
- Two basic algorithms
  - Aggregation
  - Projected holes

# Aggregated Holes

- Combine uncovered tasks with common values in some attributes
- Similar to Karnaugh maps

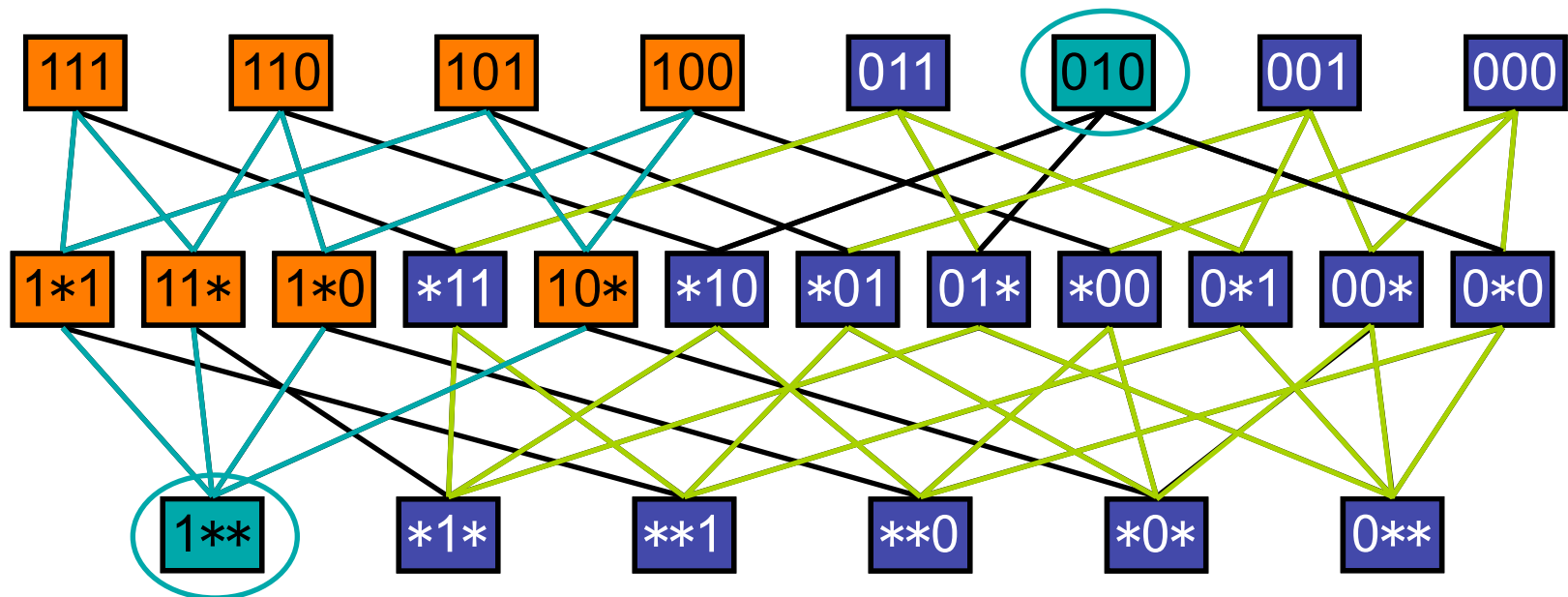


# Projected Holes

- Find holes that are **complete subspaces** of the coverage space
- Holes are in the form  $\langle q_1, q_2, \dots, q_n \rangle$ 
  - $q_i$  is either a single value or a wildcard (\*)
  - Hole dimension is the number of wildcards
  - Example:  $\langle \text{fadd}, \text{add}, *, \text{WW} \rangle$  has dimension 1
- Hole  $p$  is an ancestor of  $q$  if all the tasks in  $q$  are in  $p$ 
  - $\langle \text{fadd}, *, *, \text{WW} \rangle$  is ancestor of  $\langle \text{fadd}, \text{add}, *, \text{WW} \rangle$
- **Holes with higher dimensions usually represent larger subspaces and are more important**

# Projected Holes Algorithm

- Build layered network of all subspaces
- Recursively mark ancestors of **covered tasks**
- Loop from the bottom
  - Report unmarked nodes as holes
  - Recursively mark descendents

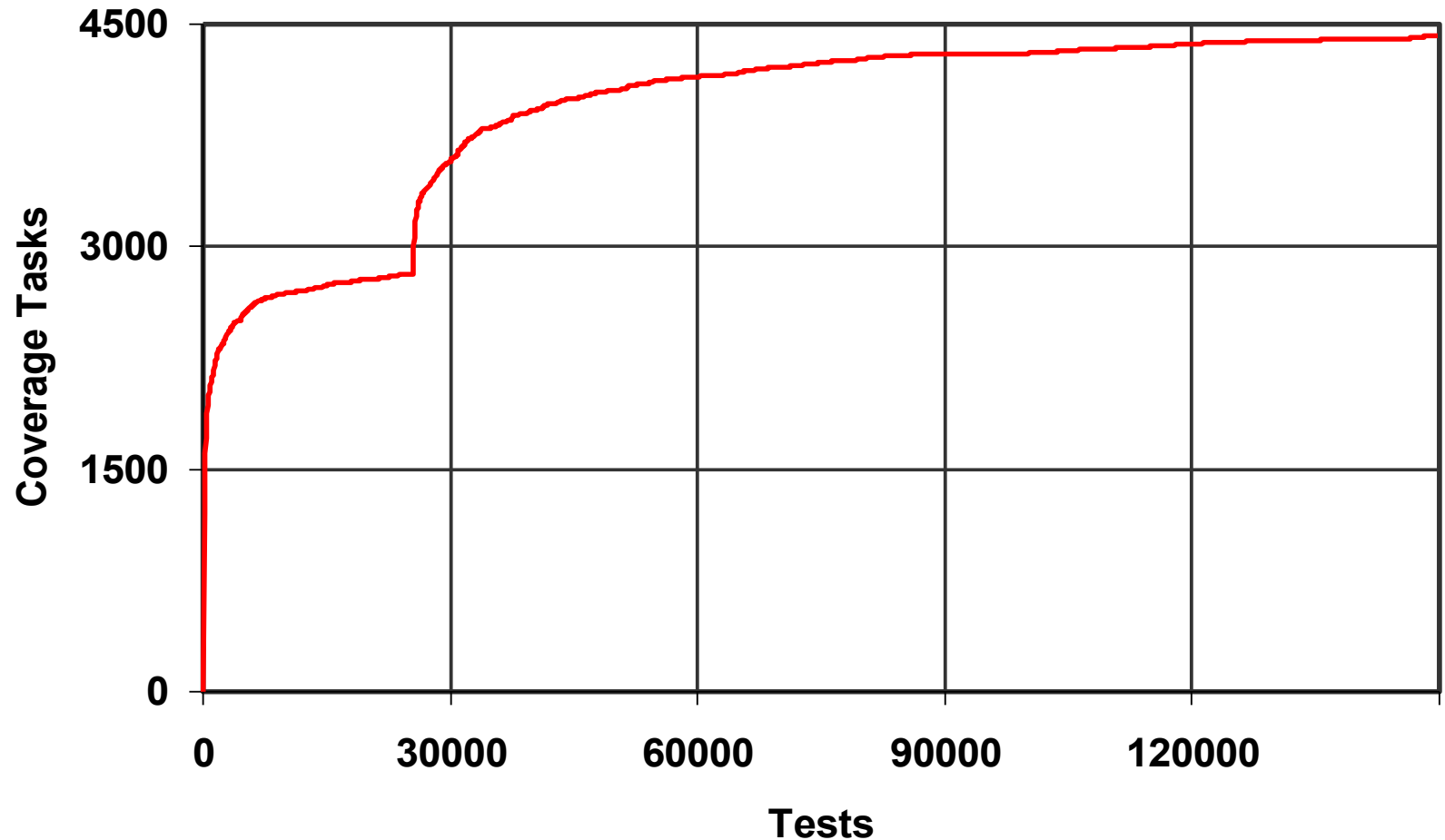


# Coverage Progress

---

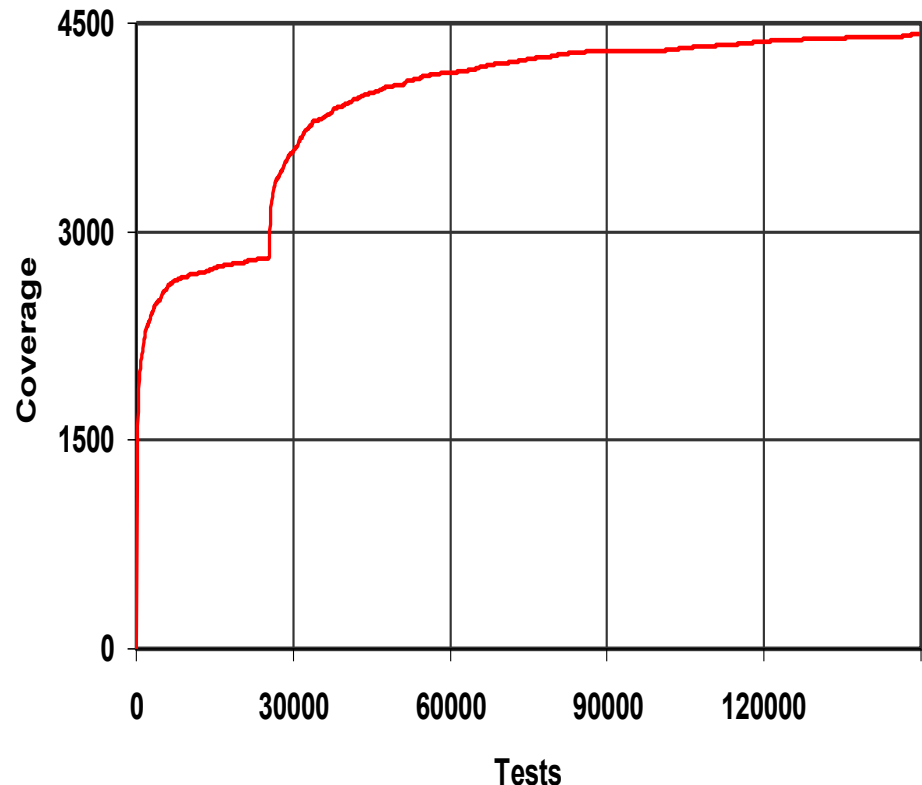
- Shows the progress of coverage over time
- Time can be measured by
  - Wall clock (or calendar) time
  - Number of tests
  - Number of simulation cycles
- Can be used on the entire coverage model or specific views of it

# Coverage Progress Example



# Progress Report Usage

- Progress report can provide a lot of information
  - How well we are progressing overall
  - What is the current progress rate
  - Are we experiencing changes in the progress rate
  - What is the expected maximal coverage
  - When it would be reached





# Using Coverage – What can go wrong?

---

- Low coverage goals
- Some coverage models are ill-suited to deal with common problems
  - Missing code
    - Use Requirements-based Methodology to overcome this!
- Generating simple tests just to cover specific uncovered tasks
  - There is merit in generating tests outside the coverage!
- Collecting coverage without analyzing and interpreting the results

WHY?

# Summary: Coverage

---

- Coverage is an important verification tool.
  - **Code** coverage: statement, path, expression
  - **Structural** coverage: FSM
  - **Functional** coverage models: story, attributes, values, restrictions
  - (**Assertion** coverage will be introduced during the lecture on Assertion-based Verification.)
- **Combination** of coverage models required in practice.
  - Code coverage alone does not mean anything!
- Verification Methodology should be **coverage driven**.