COMS30026 Design Verification

Coverage

Part I: Code Coverage

Kerstin Eder

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

Outline

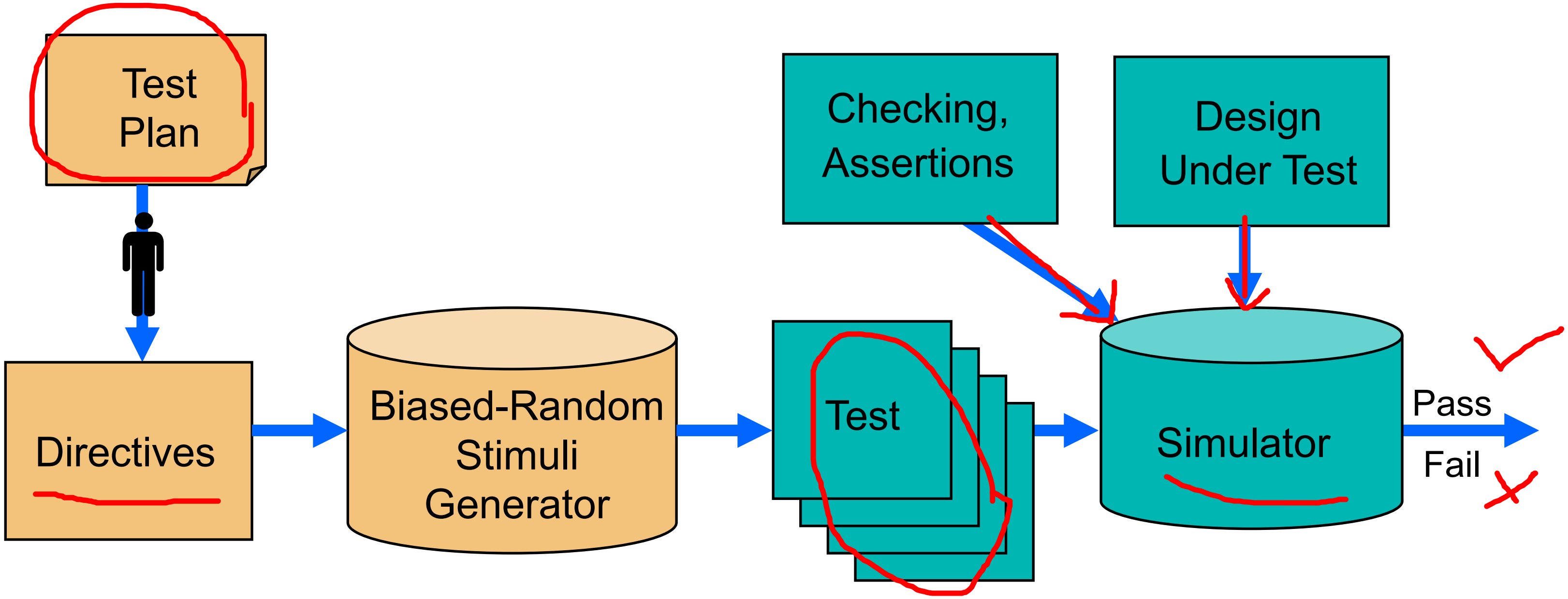
- Introduction to coverage
- Part I: Coverage Types
  - Code coverage models
  - (Structural coverage models)
- Part II: Coverage Types (continued)
  - Functional coverage models
- Part III: Coverage Analysis

Previously: Verification Tools

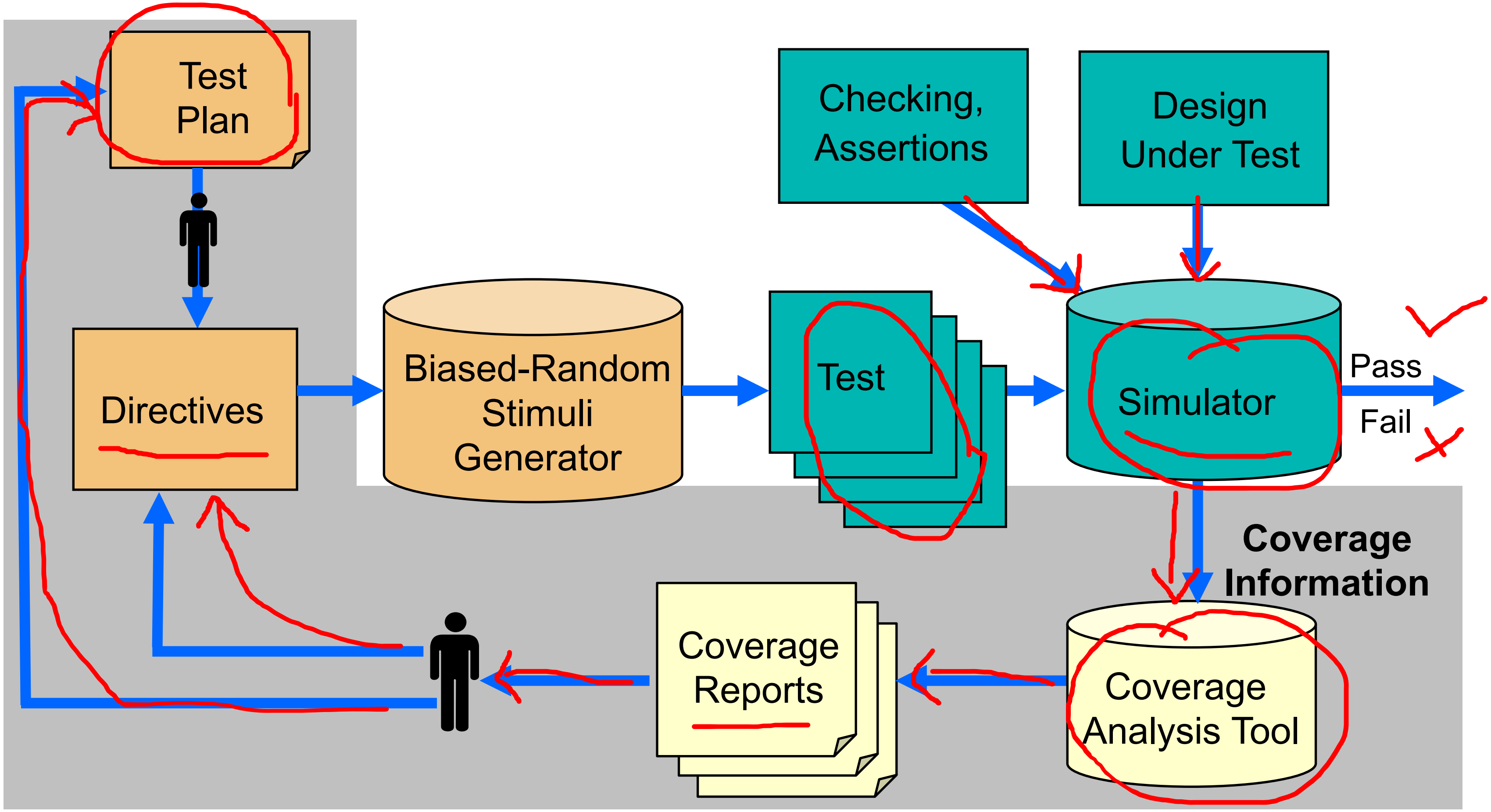
- Coverage is part of the Verification Tools.

INTRODUCTION

Simulation-based Verification Environment



Simulation-based Verification Environment



Why coverage?

- Simulation is based on limited execution samples
  - We cannot run all possible scenarios, but
  - we need to know that all (important) areas of the DUV have been exercised (and thus 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 can be used to

- Measure the "quality" of a set of tests
  - Coverage gives us an insight into what **has not been** verified!
  - Coverage completeness does not imply functional correctness of the design! **Why?** ←

Coverage can be used to

- Measure the "quality" of a set of tests
  - Coverage gives us an insight into 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

Coverage can be used to

- Measure the "quality" of a set of tests
  - Coverage gives us an insight into 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

Code Coverage - Basics

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

Code Coverage - Scope

- Code coverage can answer the question:

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

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 tried gcov?
    - No? Then, now is the right time to try it out. Please visit https://gcc.gnu.org/onlinedocs/gcc/Gcov.html and have a go.

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 tried gcov?

    - No? Then, now is the right time to try it out. Please visit https://gcc.gnu.org/onlinedocs/gcc/Gcov.html and have a go.

- **Useful for profiling:**

  – Run coverage on testbench to indicate which areas are executed most often.

  – **Gives insights on what to optimize!**

Types of Code Coverage Models

- Control flow
  - Used to determine whether the control flow of a program has been fully exercised
- Data flow
  - Used to track the flow of data in and between programs and modules
- Mutation
  - Models that can detect common bugs by mutating the code and comparing results

Control Flow Models

- Routine (function entry)
  - Each function / procedure has been called
- Function call
  - Each function has been called from every possible location
- Function return
  - Each return statement has been executed

Control Flow Models

- Routine (function entry)
  - Each function / procedure has been called
- Function call
  - Each function has been called from every possible location
- Function return
  - Each return statement has been executed
- Statement (block)
  - Each statement in the code has been executed
- Branch/Path
  - Each branch in branching statements has been taken
    - `if, switch, case, when, …`
- Expression/Condition
  - Each input in a Boolean expression (condition) has evaluated to true and also to false
    - (See further details later on MC/DC coverage)

Control Flow Models

- Routine (function entry)
  - Each function / procedure has been called
- Function call
  - Each function has been called from every possible location
- Function return
  - Each return statement has been executed
- Statement (block)
  - Each statement in the code has been executed
- Branch/Path
  - Each branch in branching statements has been taken
    - `if, switch, case, when, …`
- Expression/Condition
  - Each input in a Boolean expression (condition) has evaluated to true and also to false
    - (See further details later on MC/DC coverage)
- Loop
  - All possible numbers of iterations in (bounded) loops have been executed

Statement/Block Coverage

Measures which lines (statements) have been executed by the test 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%?

Statement/Block Coverage

Measures which lines (statements) have been executed by the test 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?
- Was it simply forgotten?

Statement/Block Coverage

Measures which lines (statements) have been executed by the test 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?
- Was it simply forgotten?
- Is it a condition that can never occur?
  - (Dead code might be “ok”!) WHEN & WHY?

Path/Branch Coverage

Measures all possible ways to execute a sequence of statements.

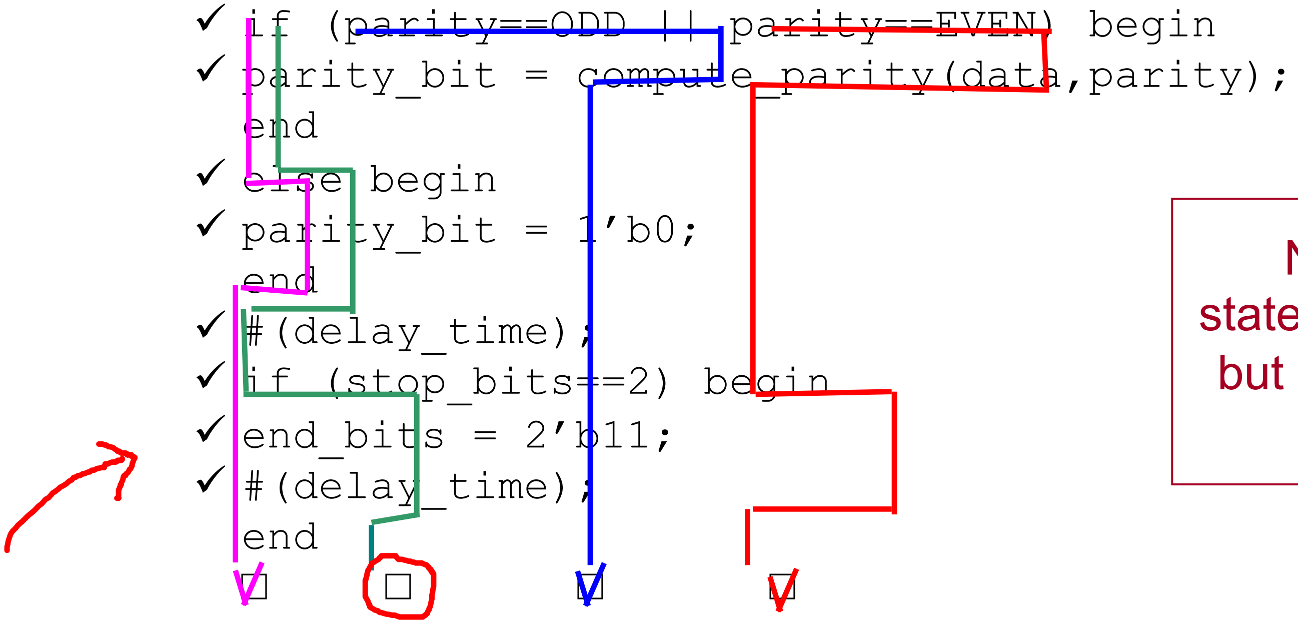
- Have all branches or execution paths been taken?
- How many execution paths?

```
✓ 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
```

Path/Branch Coverage

Measures all possible ways to execute a sequence of statements.

- Have all branches or execution paths been 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! ✓

Path/Branch Coverage

Measures all possible ways to execute a sequence of statements.

- Have all branches or execution paths been 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!
  - Consider using ignore tags!

Expression/Condition Coverage

Measures the various ways Boolean expressions and subexpressions can be 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
```

Expression/Condition Coverage

Measures the various ways Boolean expressions and subexpressions can be 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
```

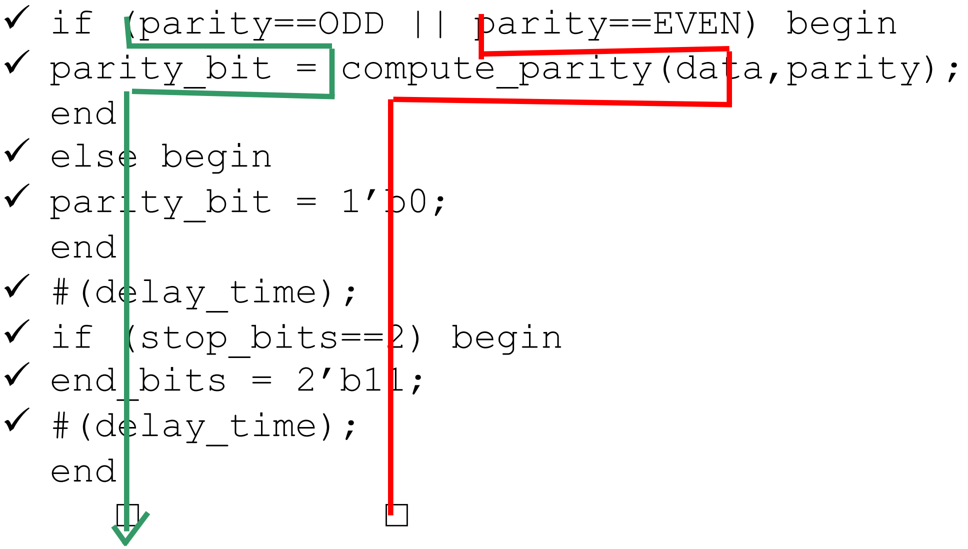
Note: Only 50% expression coverage!

- **Analysis:** Understand WHY part of an expression has not been covered

Expression/Condition Coverage

Measures the various ways Boolean expressions and subexpressions can be 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, used in certification!) ☺

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

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

Terminology:

The literals/inputs in a Boolean expression are termed **conditions.**

The output of a Boolean expression is termed **decision**.

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

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

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

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

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 of the number of test cases 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 || b` vectors TF, FT and FF satisfy this requirement

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 || b` vectors TF, FT and FF satisfy this requirement

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 || 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).**

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

```
process (a, b)
begin
   s <= a  + b;
end process

process (clk)
begin
   if (reset)
      a <= 0; b <= 0;
   else
      a <= in1; b <= in2;
   end if
end process
```

Data Flow Models

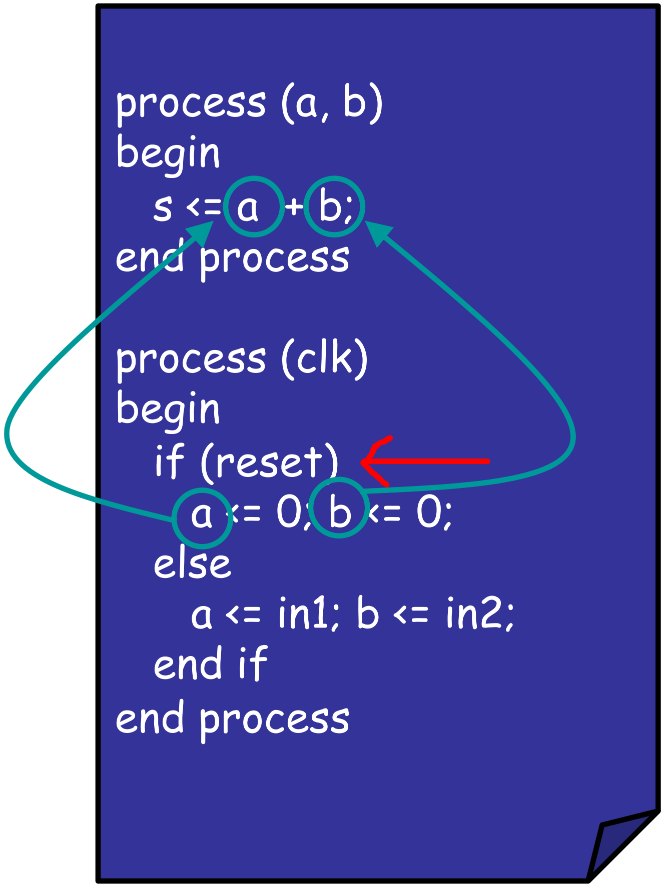
- Coverage models that are based on flow of data during execution
- Each coverage task has two attributes
  - **Define** – where a value is assigned to a variable (signal, register, …)
  - **Use** – where the value is being used
- Types of dataflow models
  - C-Use – Computational use
  - P-Use – Predicate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

```
process (a, b)
begin
   s <= a  + b;
end process

process (clk)
begin
   if (reset)
      a <= 0; b <= 0;
   else
      a <= in1; b <= in2;
   end if
end process
```

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

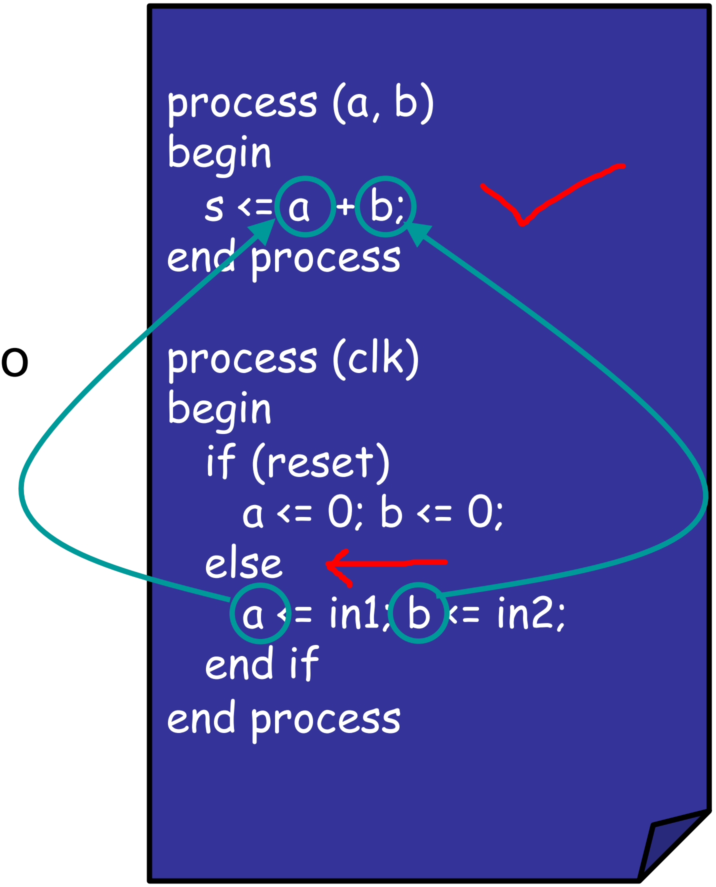


```
process (a, b)
begin
   s <= a + b;
end process

process (clk)
begin
   if (reset)
      a <= 0; b <= 0;
   else
      a <= in1; b <= in2;
   end if
end process
```

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



```
process (a, b)
begin
  s <= a + b;
end process

process (clk)
begin
  if (reset)
    a <= 0; b <= 0;
  else
    a <= in1; b <= in2;
  end if
end process
```

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

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

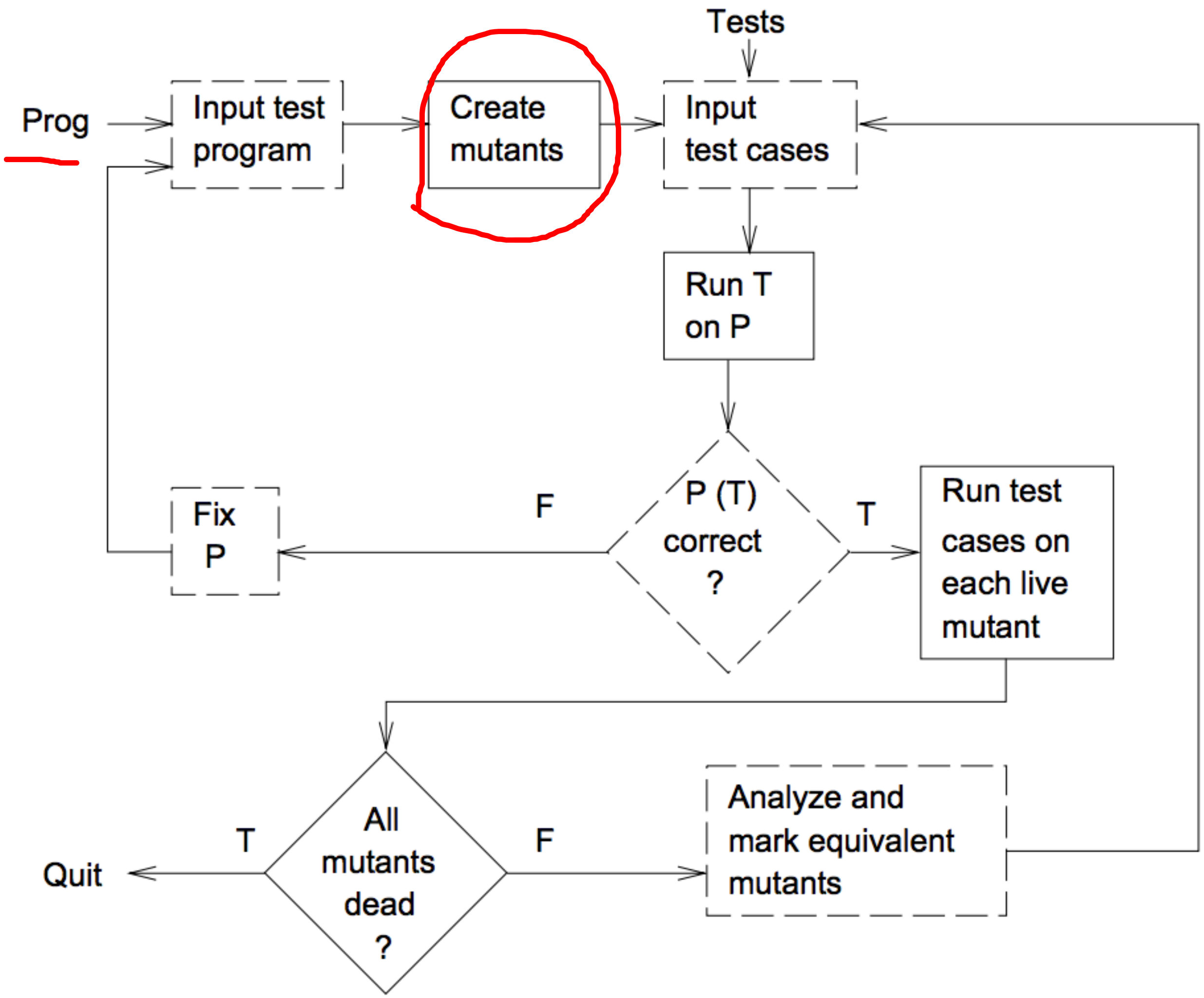


Figure 1: **Traditional Mutation Testing Process. Solid boxes represent steps that are automated and dashed boxes represent steps that are manual.**

A. Jefferson Offutt and Ronald H. Untch. 2001. Mutation 2000: uniting the orthogonal. Mutation testing for the new century. Kluwer Academic Publishers, USA, 34–44.

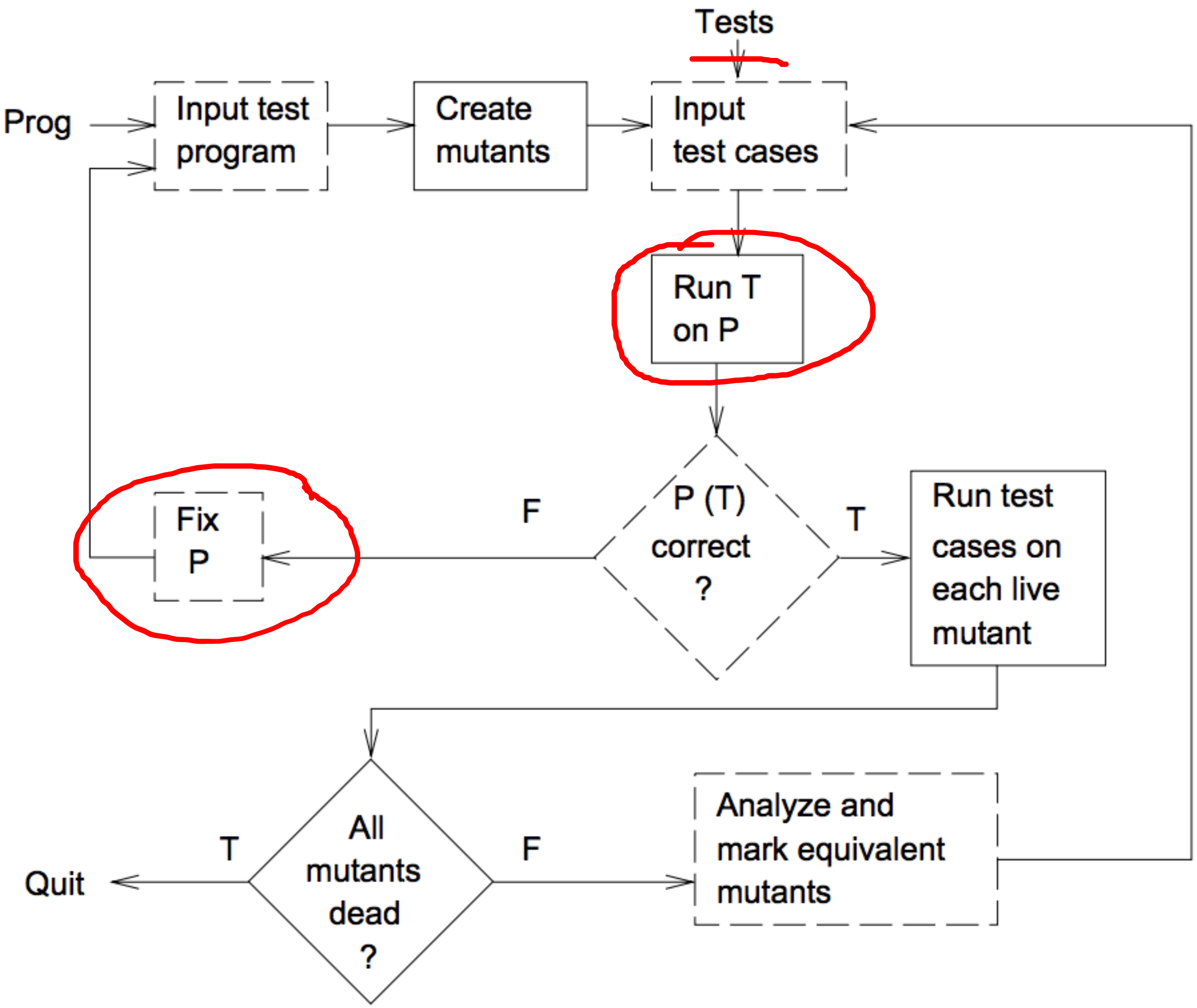


Figure 1: **Traditional Mutation Testing Process. Solid boxes represent steps that are automated and dashed boxes represent steps that are manual.**

A. Jefferson Offutt and Ronald H. Untch. 2001. Mutation 2000: uniting the orthogonal. Mutation testing for the new century. Kluwer Academic Publishers, USA, 34–44.

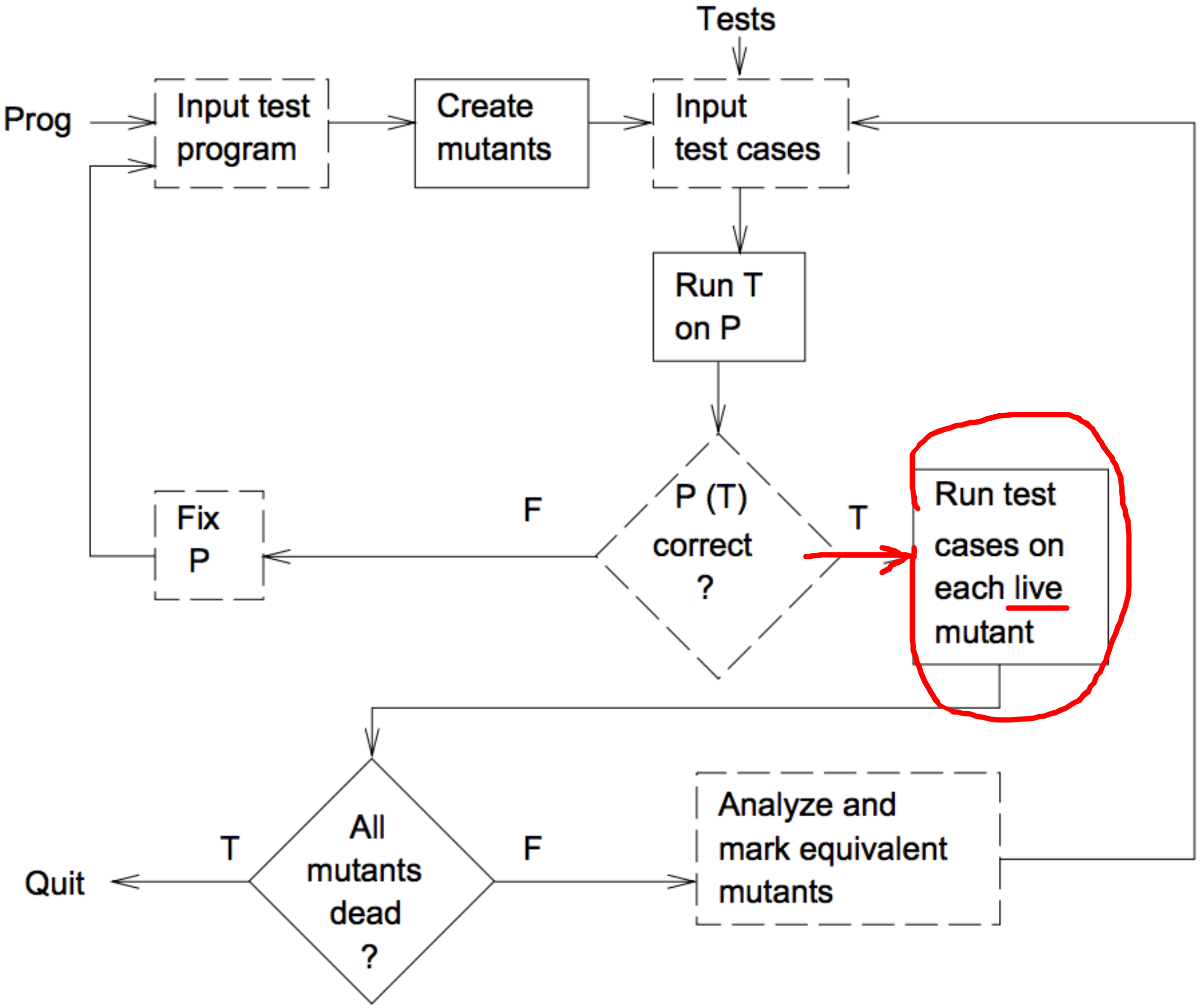


Figure 1: **Traditional Mutation Testing Process. Solid boxes represent steps that are automated and dashed boxes represent steps that are manual.**

A. Jefferson Offutt and Ronald H. Untch. 2001. Mutation 2000: uniting the orthogonal. Mutation testing for the new century. Kluwer Academic Publishers, USA, 34–44.

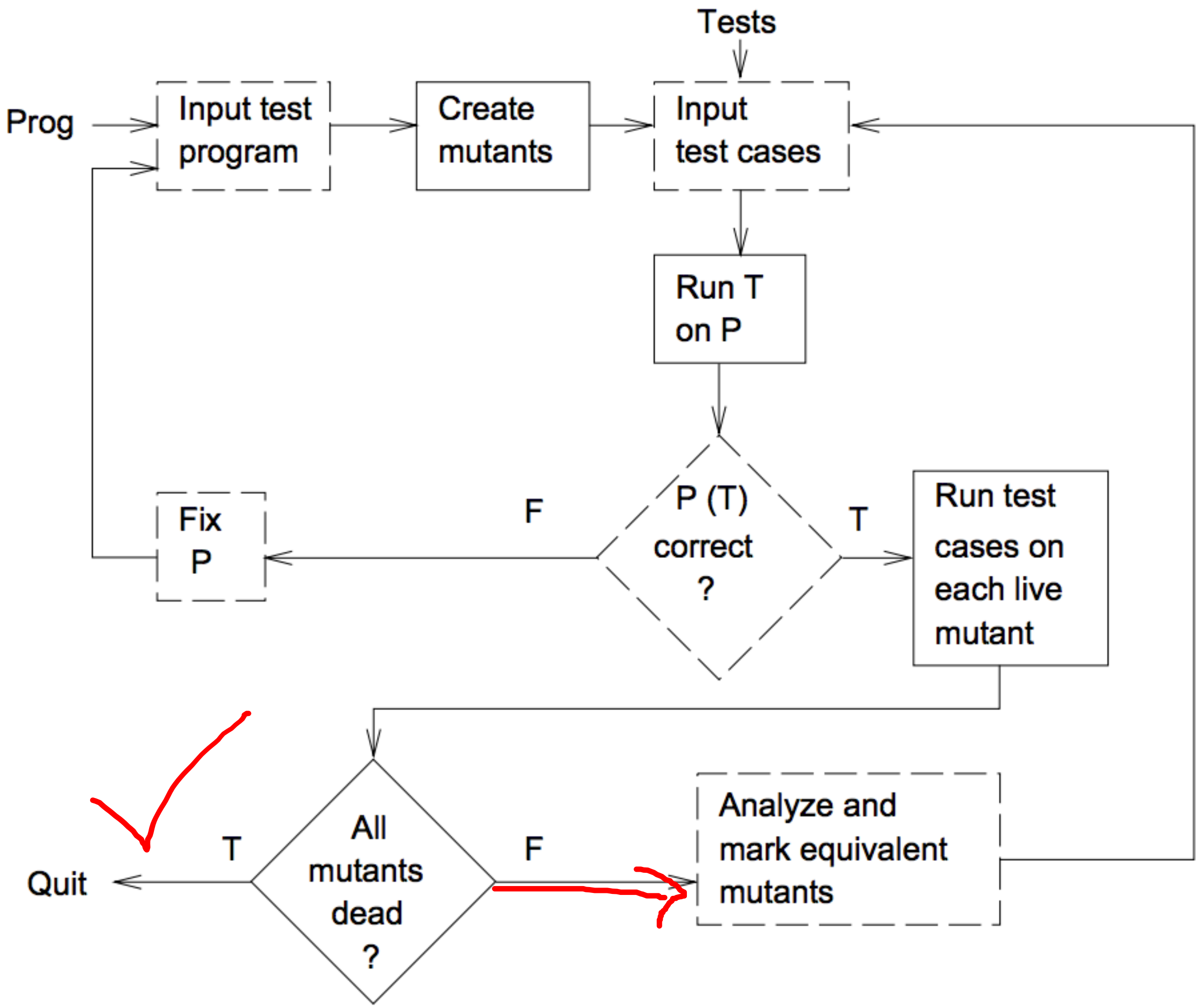


Figure 1: **Traditional Mutation Testing Process. Solid boxes represent steps that are automated and dashed boxes represent steps that are manual.**

A. Jefferson Offutt and Ronald H. Untch. 2001. Mutation 2000: uniting the orthogonal. Mutation testing for the new century. Kluwer Academic Publishers, USA, 34–44.

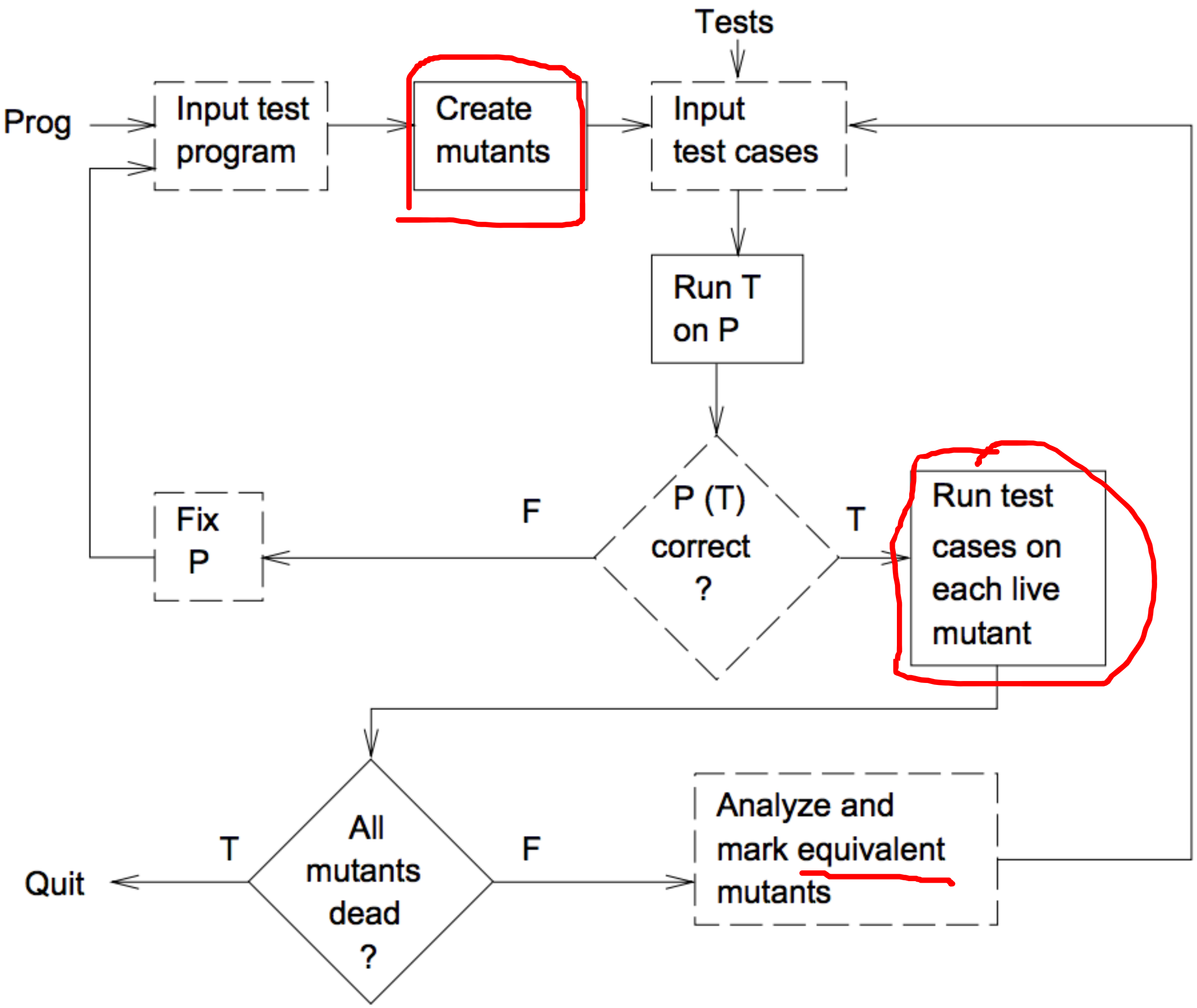


Figure 1: **Traditional Mutation Testing Process. Solid boxes represent steps that are automated and dashed boxes represent steps that are manual.**

A. Jefferson Offutt and Ronald H. Untch. 2001. Mutation 2000: uniting the orthogonal. Mutation testing for the new century. Kluwer Academic Publishers, USA, 34–44.

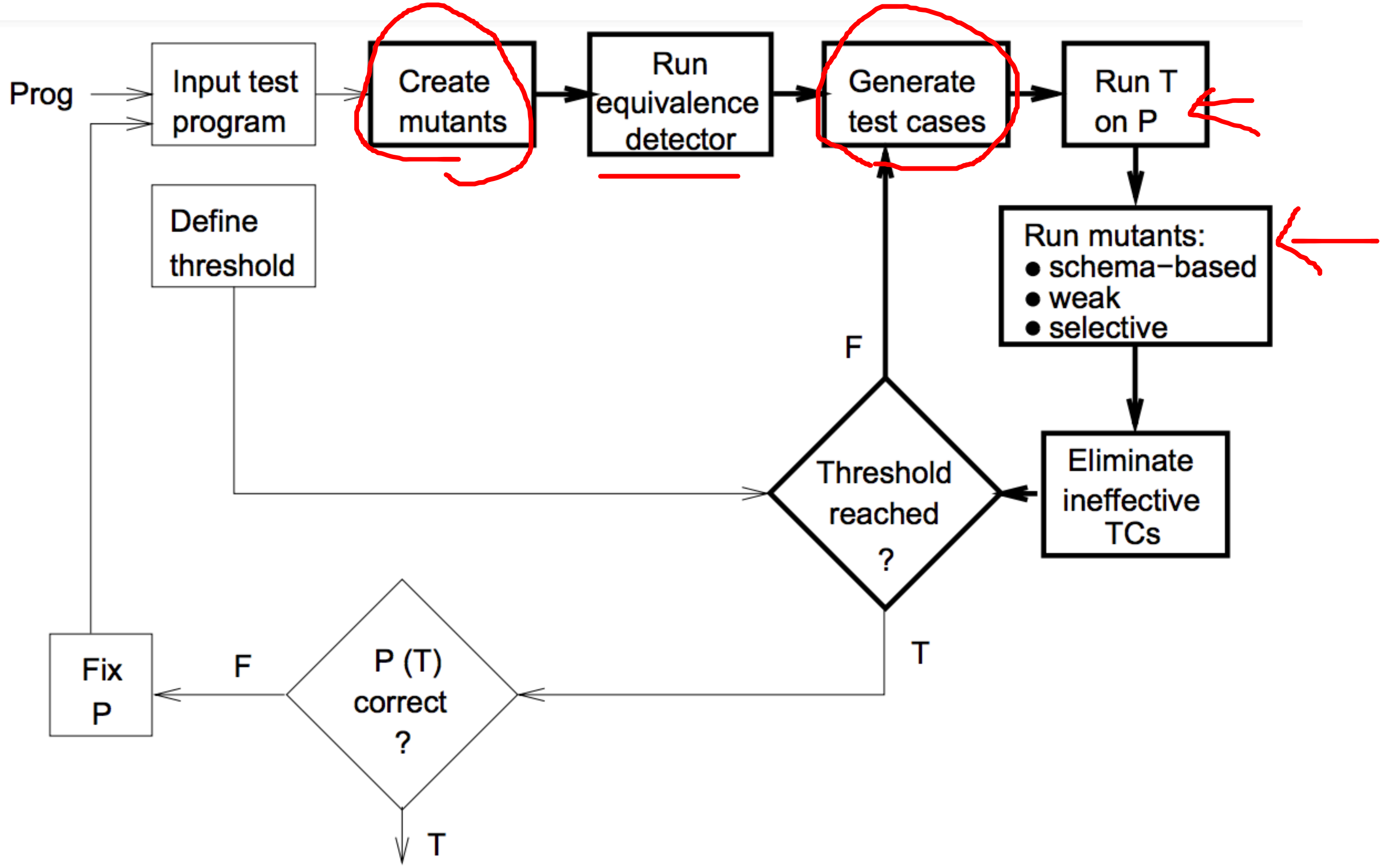


Figure 2: **New Mutation Testing Process. Bold boxes represent steps that are automated; remaining boxes represent steps that are manual.**

A. Jefferson Offutt and Ronald H. Untch. 2001. Mutation 2000: uniting the orthogonal. Mutation testing for the new century. Kluwer Academic Publishers, USA, 34–44.

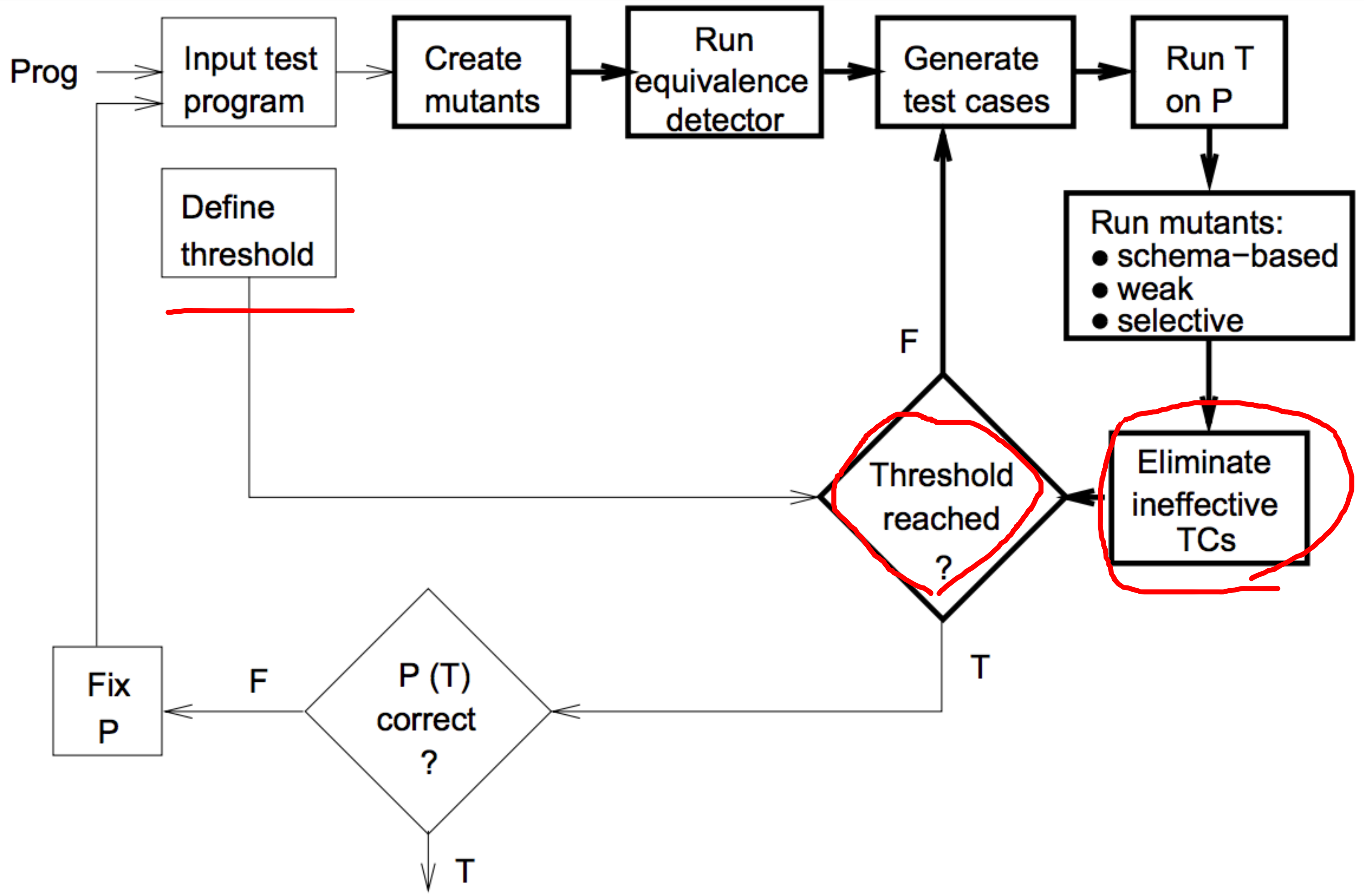


Figure 2: **New Mutation Testing Process. Bold boxes represent steps that are automated; remaining boxes represent steps that are manual.**

A. Jefferson Offutt and Ronald H. Untch. 2001. Mutation 2000: uniting the orthogonal. Mutation testing for the new century. Kluwer Academic Publishers, USA, 34–44.

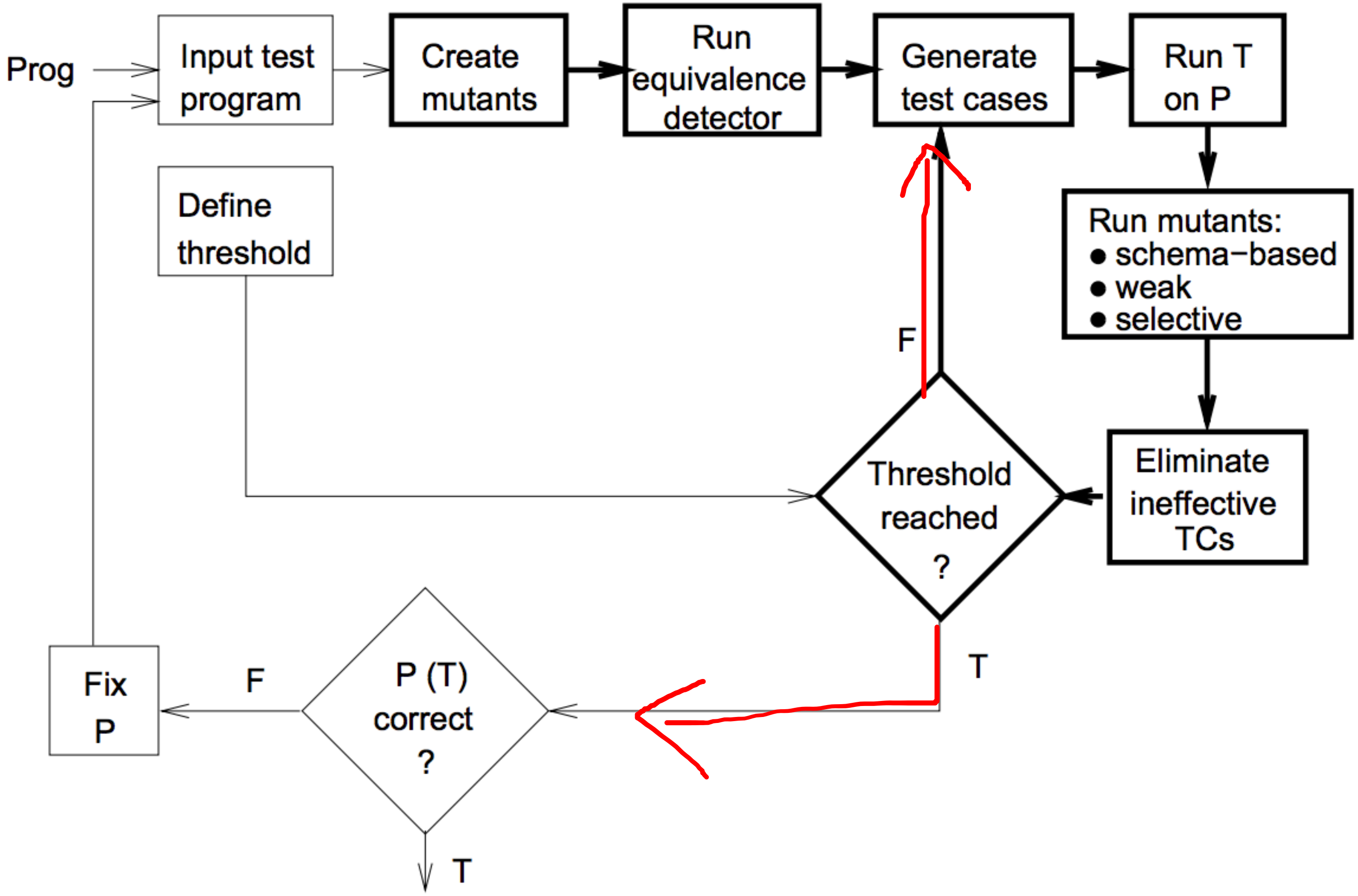


Figure 2: **New Mutation Testing Process. Bold boxes represent steps that are automated; remaining boxes represent steps that are manual.**

A. Jefferson Offutt and Ronald H. Untch. 2001. Mutation 2000: uniting the orthogonal. Mutation testing for the new century. Kluwer Academic Publishers, USA, 34–44.

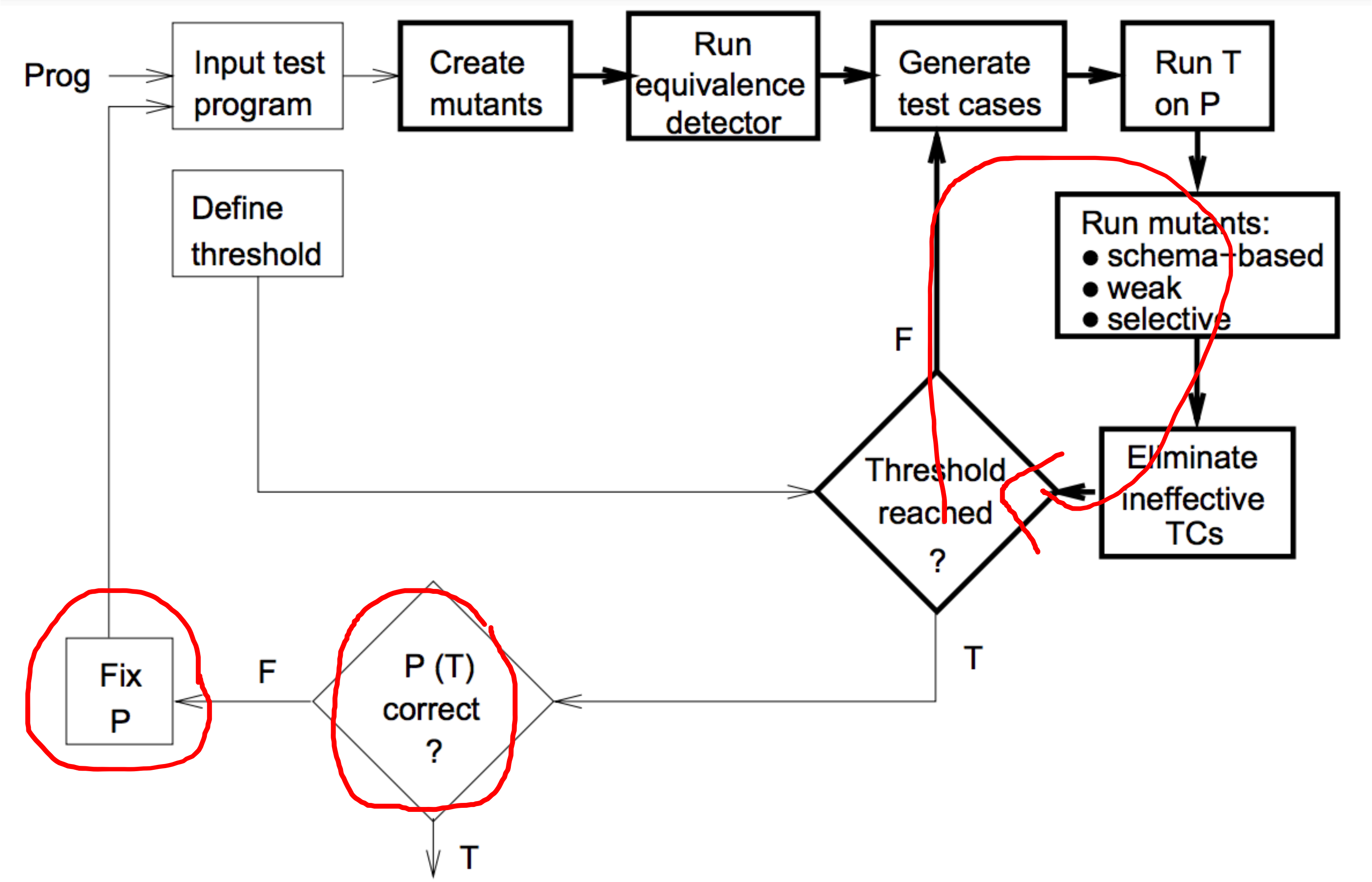


Figure 2: **New Mutation Testing Process. Bold boxes represent steps that are automated; remaining boxes represent steps that are manual.**

A. Jefferson Offutt and Ronald H. Untch. 2001. Mutation 2000: uniting the orthogona Mutation testing for the new century. Kluwer Academic Publishers, USA, 34–44.

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

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-level testing.

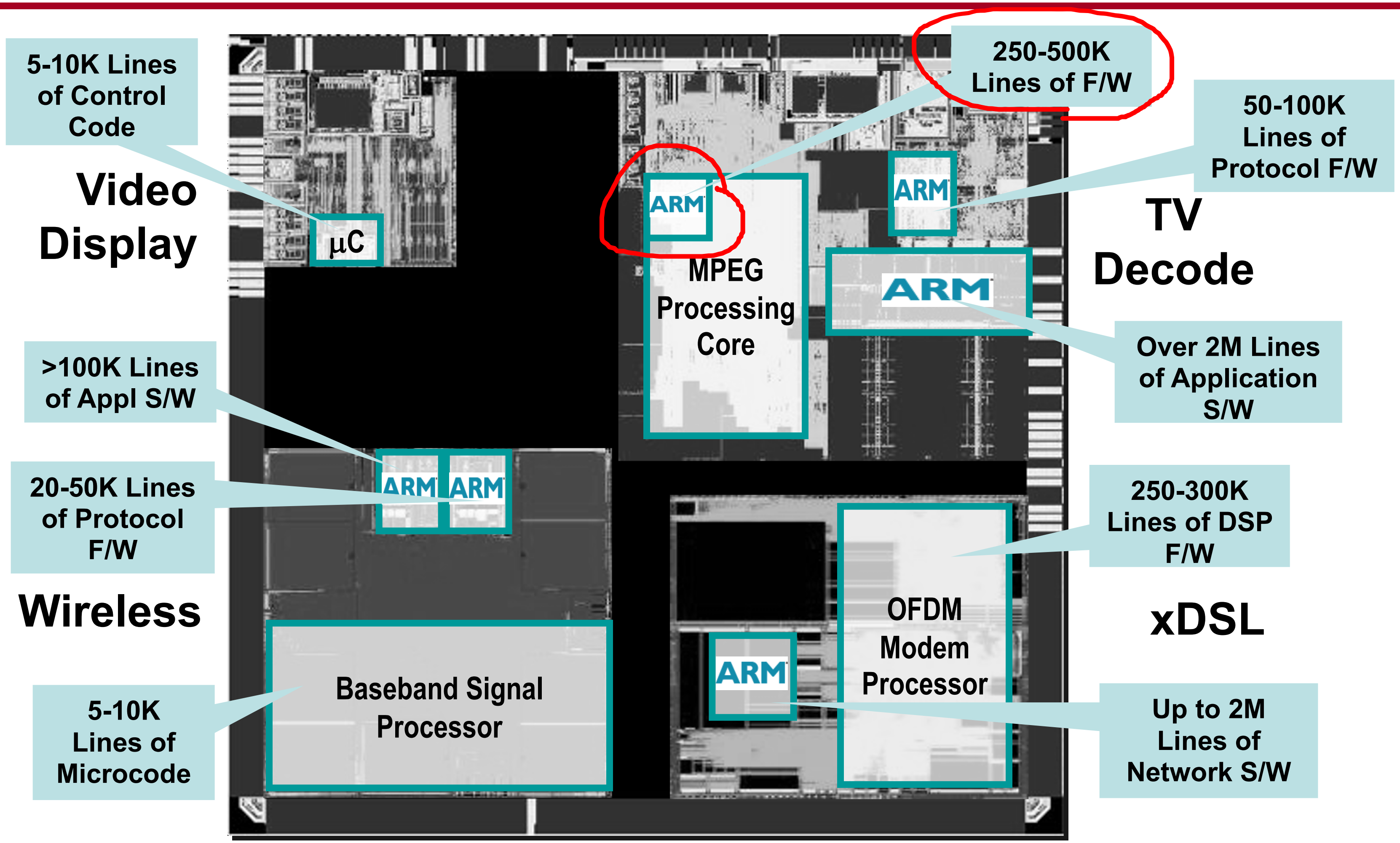
Code Coverage Strategy

- 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-level testing.
- Generally, verification plans include a 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.

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-level testing.
- Generally, verification plans include a 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%.

Increasing Design Complexity



5-10K Lines of Control Code

Video Display

>100K Lines of Appl S/W

20-50K Lines of Protocol F/W

Wireless

5-10K Lines of Microcode

250-500K Lines of F/W

50-100K Lines of Protocol F/W

TV Decode

Over 2M Lines of Application S/W

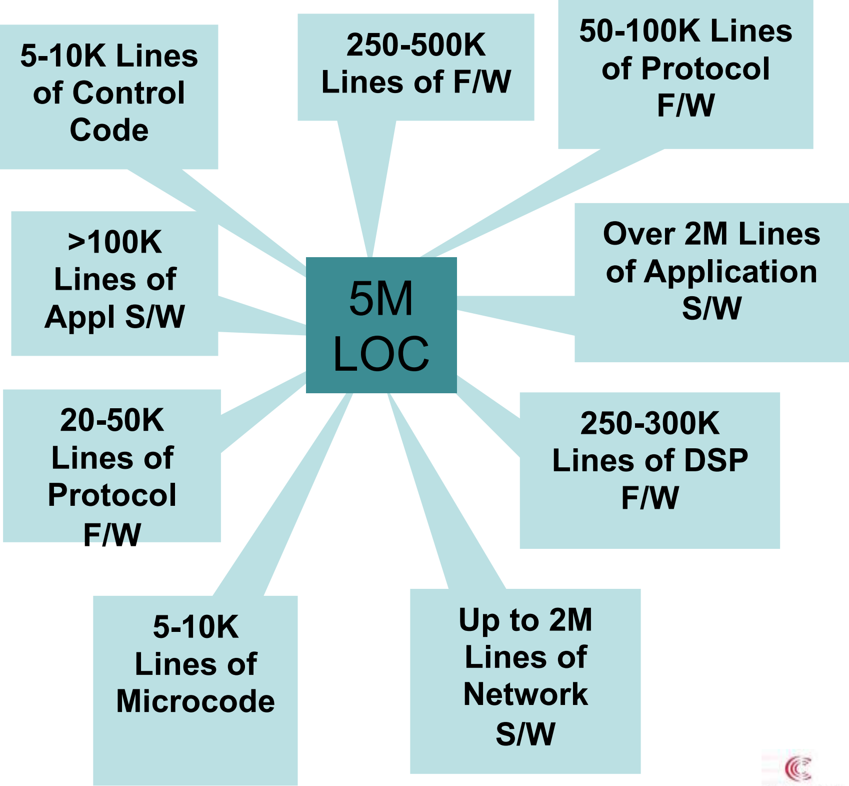
250-300K Lines of DSP F/W

xDSL

Up to 2M Lines of Network S/W

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

Increasing Design Complexity



5M LOC

5-10K Lines of Control Code

250-500K Lines of F/W

50-100K Lines of Protocol F/W

>100K Lines of Appl S/W

Over 2M Lines of Application S/W

20-50K Lines of Protocol F/W

250-300K Lines of DSP F/W

5-10K Lines of Microcode

Up to 2M Lines of Network S/W

LOC count:

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

TOTAL: ~5M LOC

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

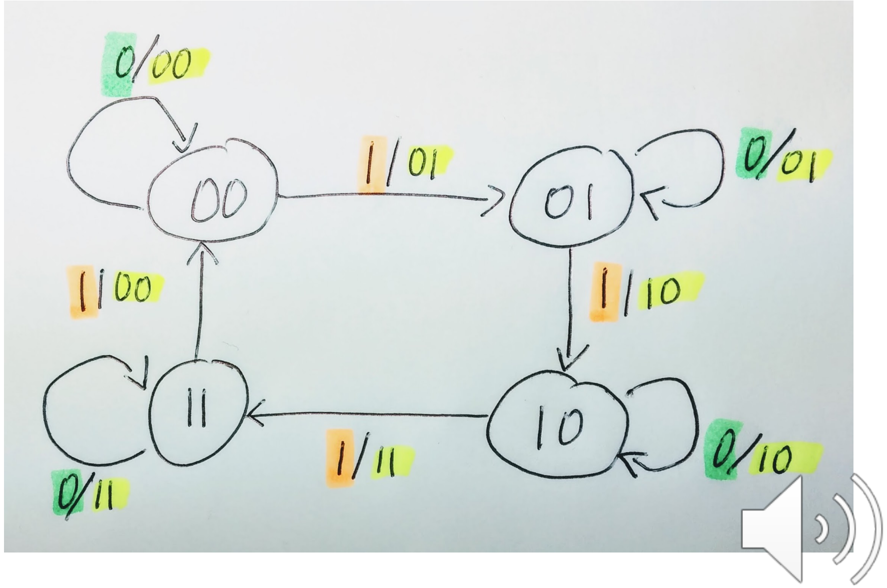
STRUCTURAL COVERAGE

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 coverage models
  - Identify and declare illegal events

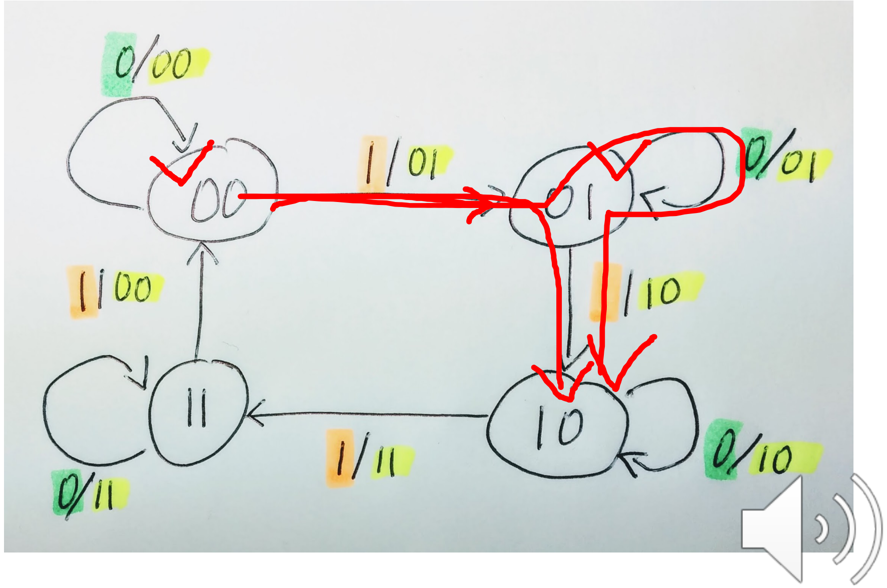
State-Machine Coverage

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

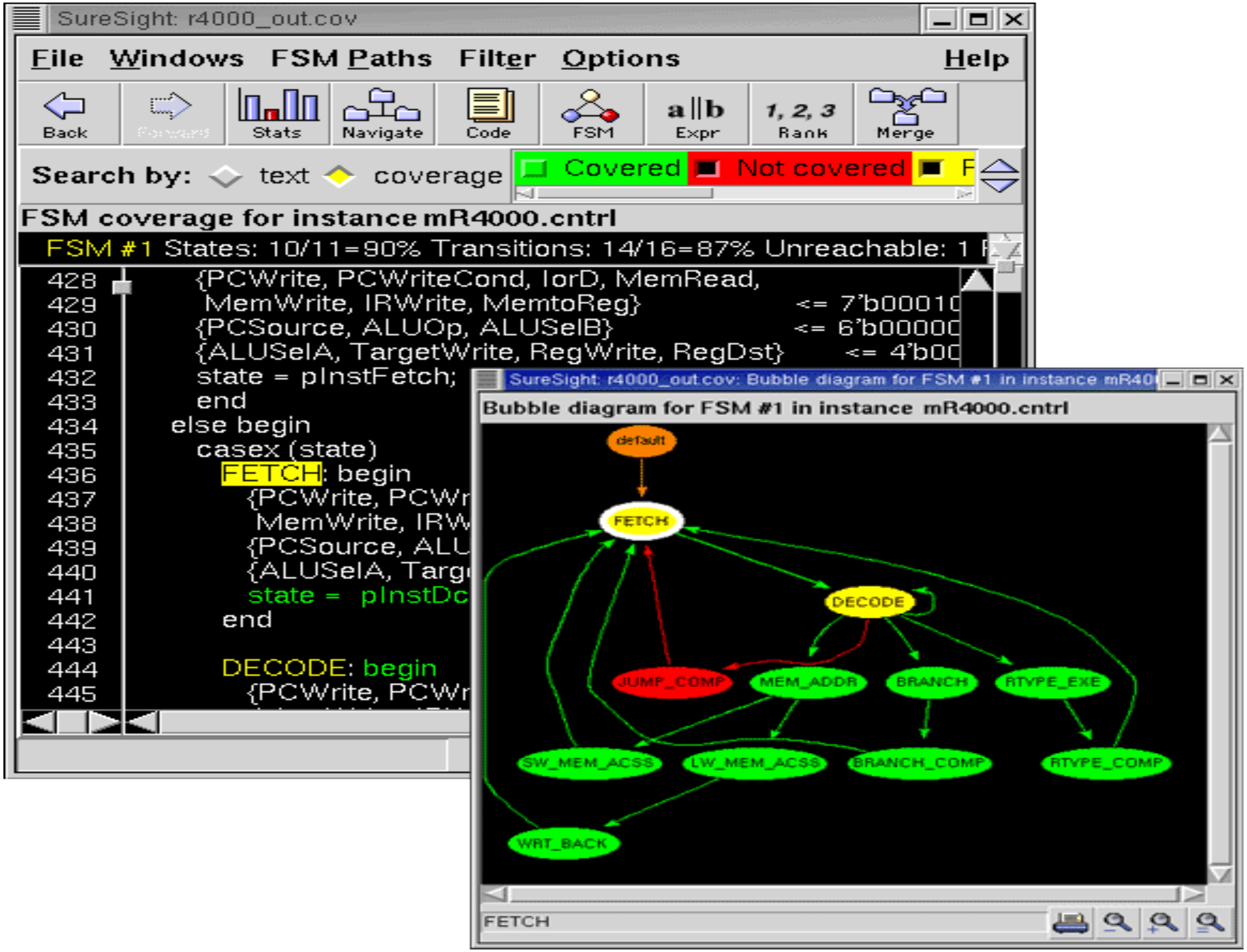


State-Machine Coverage

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



FSM Coverage Report



FSM coverage for instance mR4000.cntrl

FSM #1 States: 10/11=90% Transitions: 14/16=87% Unreachable: 1 F

```
        {PCWrite, PCWriteCond, IorD, MemRead,
         MemWrite, IRWrite, MemtoReg}             <= 7'b00010
        {PCSource, ALUOp, ALUSelB}                <= 6'b00000
        {ALUSelA, TargetWrite, RegWrite, RegDst}     <= 4'b00
        state = pInstFetch;
        end
     else begin
        casex (state)
          FETCH: begin
            {PCWrite, PCWr
             MemWrite, IRW
            {PCSource, ALU
            {ALUSelA, Targ
            state =  pInstDc
          end

          DECODE: begin
            {PCWrite, PCWr
```

Bubble diagram for FSM #1 in instance mR4000.cntrl

Code Coverage - Limitations

- Coverage questions not answered by code coverage
  - Did every instruction take every exception?
  - Did two instructions access a specific register at the same time?
  - How many times did a cache miss take more than 10 cycles?
  - …(and many more)
  - → Does the implementation cover the functionality specified? [Need RBT!]

Code Coverage - Limitations

- Coverage questions not answered by code coverage
  - Did every instruction take every exception?
  - Did two instructions access a specific register at the same time?
  - How many times did a cache miss take more than 10 cycles?
  - …(and many more)
  - Does the implementation cover the functionality specified? [Need RBT!]
- Code coverage only 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 permit any conclusions about the functional correctness of the code, nor does it help us understand whether all the functionality was covered!

So, 100% code coverage does not mean very much. ☹

- We need another form of coverage!