

WHITE-BOX TESTING

Week #7

White Box testing

Tester can see the code

Pseudonyms: Glass Box, Clear Box, Crystal Box, Structural Testing

Provides much better code coverage than Black Box testing

The following are visible to the tester:

- *Boundaries*
- *Control flow and Data flow*
- *Complexity*

Examples of White Box Techniques

Dynamic Testing

- Structural Coverage (statement, branch/ decision, condition)
- Control Flow analysis (path coverage)
- Data Flow analysis (data usage coverage)
- Code Profiling/Instrumentation

Static Testing

- Structured Evaluations
- Static Analysis

Static Testing

“Static”: Not “Dynamic”, i.e., you don’t execute the code

This includes some testing of documentation

Two forms:

Structured Evaluations

Static Analysis

Static Analysis

Static Analysis uses Tools applies more to the software code than to documentation

Examples

Compiling

Compliance with Conventions and Standards

Examination of the Data Flow graph

Examination of the Control Flow graph

Metrics (e.g. Cyclomatic Complexity)

Structural Coverage

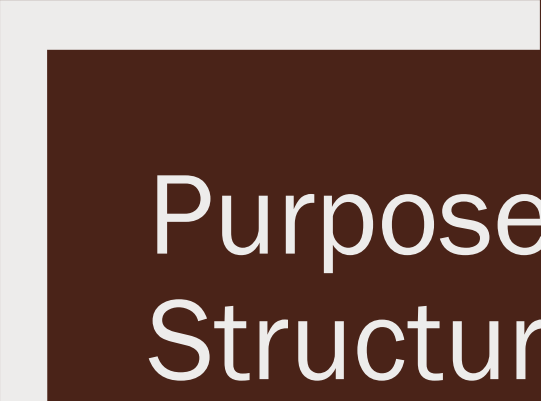
- Coverage is a metric, not a method
- The other broad coverage category is Requirements Coverage
- Code that is implemented without being linked to requirements may not be exercised by requirements-based tests

Rationale for Structural Coverage

Statistical approaches to quality assurance, which work well for physical devices, do not apply to software

Measure of the completeness of testing

- Using code coverage metrics to cross-check the Functional tests is a good idea
- Can be hard to implement



Purpose of Structural Coverage

Unit Test use: Exercise the code; look for unintended functionality, incorrect behavior, unreachable code.

Requirements Test use: Reveal code that has not been executed and which thus needs new test cases.

Structural Coverage can improve the tests as much as it improves the code.

Covering Logic Expressions in the Software

- Logic expressions show up in many situations
- As just one example, covering logic expressions is required by the US Federal Aviation Administration for safety critical software
- Logical expressions occur in decisions or branch points in the software
- Tests are intended to choose some subset of the total number of truth assignments possible in the expressions

Types of Coverage

Statement
Coverage

Branch (or
Decision) Coverage

Condition Coverage

Multiple Condition
Coverage

Modified
Condition/Decision
Coverage
("MC/DC")

Path Coverage

Points to Remember



Verifying these types of coverage is all well and good, but it serves no purpose unless you're also verifying that, in each case, you get the program results that you want.



If your coverage tool says you got 100% coverage, but every single output was incorrect, then you have a problem!

Conditions and clauses

- A *condition* is an expression that evaluates to a Boolean value
- Conditions can contain
 - *Boolean variables*
 - *Relational operators: >, <, ==, >=, <=, !=*
 - *Boolean function calls*
 - *Boolean (logical) operators: AND, NOT, OR, etc.*
- A *clause* is a condition with no Boolean operators –this an “atomic (partial) condition”
 - *Each clause evaluates to either T or F.*
- A branch is a pathway out of a decision

Example

A Sample Condition...

$$((a < b) \vee f(z)) \wedge D \wedge (m \geq n * o)$$

... with four clauses:

1. $(a < b)$ – relational expression
2. $f(z)$ – Boolean-valued function
3. D – Boolean variable
4. $(m \geq n * o)$ – relational expression (with an arithmetical operator)

Logic Coverage Criteria Definitions

Statement or line coverage: Execute every line of code at least once [each node in the program graph]

Branch coverage: Execute each branch of every decision [Hence: “Decision coverage”]

Condition coverage: Execute every decision with each possible condition for each clause.

Multiple condition coverage: Execute each decision with all combinations of conditions for the clauses

Path coverage: Execute each path in the program graph

A Coverage Example

IF ($A < B$ AND $C = 5$) THEN

 DO {something};

SET $D = 5$;

Test cases you might think of:

- a. $A < B$, $C = 5$ [or constraint “(T, T)”]
- b. $A < B$, $C \neq 5$ [or constraint “(T, F)”]
- c. $A \geq B$, $C = 5$ [or constraint “(F, T)”]
- d. $A \geq B$, $C \neq 5$ [or constraint “(F, F)”]

A Coverage Example

```
IF ( A < B AND C = 5) THEN  
    DO {something};
```

```
SET D = 5;
```

Associated test cases:

(a) $A < B$, $C = 5$

(b) $A < B$, $C \neq 5$

(c) $A \geq B$, $C = 5$

(d) $A \geq B$, $C \neq 5$

Statement coverage (1)

Branch coverage (2)

Condition coverage (2)

Multiple condition coverage: All 4

Example

[Inadequacy of Statement Coverage]

A C/C++ code fragment:

```
int* p = NULL;  
if (some condition)  
    p = &variable;  
*p = 123;
```

Example

[Inadequacy of Branch Coverage]

```
if (A && (B || C))  
    do-this;  
else  
    do-that;
```

A = FALSE executes do-that

A = B = TRUE executes do-this

We get Branch Coverage without even looking at C.

Example

[Inadequacy of Condition Coverage]

If Not (A or B) then C

Setting A = True, B = False, then A = False and B = True, satisfies Condition Coverage, but statement block C never gets executed;

i.e. we get neither statement coverage nor branch coverage!

[A has taken on the values T and F, and B has taken on the values T and F – that's all]

The “Obvious” Solution: Multiple Condition Coverage

- Also called Combinatorial Coverage
- Requires that each possible combination of inputs be tested for each decision.
- Example: “if (A or B) then C” requires 4 test cases:
 - A = True, B = True
 - A = True, B = False
 - A = False, B = True
 - A = False, B = False

Multiple Condition Coverage

Multiple Condition Coverage does indeed give us Statement, Branch, and Condition Coverage.

The problem: For n conditions, 2^n test cases are needed, which grows exponentially with n

Controlling the Combinatorial Explosion

Modified Condition/Decision Coverage (MC/DC), or Condition Determination, of n conditions requires only $n + 1$ test cases.

A document on the topic:

“A Practical Tutorial on Modified Condition/Decision Coverage”

<http://shemesh.larc.nasa.gov/fm/papers/Hayhurst-2001-tm210876-MCDC.pdf>

Modified Condition/Decision Coverage

- Requires that each condition be shown to independently affect the outcome of a decision, by varying one clause and leaving the other clauses fixed.
- Generally, for n conditions, MC/DC requires only $n + 1$ test cases
- Example: if (A or B) then C

3 test cases: A = True, B = False (A rules)

A = False, B = True (B rules)

A = False, B = False (the *false* outcome)

Some MC/DC Examples

■ To test If (A OR B):

A:	T	F	F
B:	F	T	F
Result:	T	T	F

■ To test If (A AND B)

A:	F	T	T
B:	T	F	T
Result:	F	F	T

MC/DC Example

Which test cases below show each condition independently affecting the outcome? (How many should there be?)

Decision: X or Y or Z

<u>X</u>	<u>Y</u>	<u>Z</u>	<u>Result</u>
F	F	F	F
T	F	F	T
F	T	F	T
T	T	F	T
F	F	T	T
T	F	T	T
F	T	T	T
T	T	T	T

MC/DC Example

Which test cases below show each condition independently affecting the outcome?

Decision: X and Y and Z

X	Y	Z	Result
F	F	F	F
T	F	F	F
F	T	F	F
T	T	F	F
F	F	T	F
T	F	T	F
F	T	T	F
T	T	T	T

Additional Notes

- Path Coverage guarantees only Statement and Branch coverage
- Not even Multiple Condition Coverage guarantees Path Coverage
- MC/DC (and the other such coverage criteria) affect all decisions, not just branch points:
 - *“If ... Then” statements*
 - *“If ... Then ... Else” statements*
 - *Case statements*
 - *loop while and loop until controls*
 - *Assignment statements: $A = (B \text{ or } C)$*

Besides covering all these logic paths, don't forget to verify that the result in each case is what you actually want!

CONTROL FLOW ANALYSIS/PATH COVERAGE

Why: If you have coded a path in a program, then that path should be exercised by *you* before releasing it to an unsuspecting, naïve world.

Take-away: Know what a Program Graph is and how to calculate and test the paths in that graph.

Path Analysis

- This is part of Structural Testing
- *Program Graph*: A directed graph in which the nodes are statements (or statement fragments) and the edges represent control flow.
- An edge connects one node to another provided the destination node can be executed immediately following the source node.

A Process for Path Analysis

- Construct the Program Graph for the program module under test
- Condense it to a DD-Path Graph (“DD” = “Decision-to-Decision”)
- Count and list the syntactical paths
- Construct the rules for various nodes
- Count and list the semantic (feasible) paths using the rules
- Design the test cases to exercise those paths
- References:
 - *The construction of a Control Flow graph is right at the beginning of Section 5.2.1 (Statement Testing and Coverage).*
 - *The text’s example of Path Coverage begins halfway through Section 5.2.3 (Test of Conditions).*

The Triangle Program

In order for 3 integers a , b , and c to be the sides of a triangle, we must have

$$c1 \quad a + b > c$$

$$c2 \quad a + c > b$$

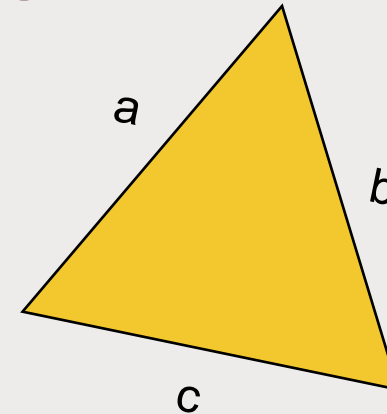
$$c3 \quad b + c > a$$

A triangle is:

Equilateral if all 3 sides are equal

Isosceles if (exactly) 2 sides are equal

Scalene if no two sides are equal



The Triangle Program cont'd

- The Triangle Program reads in 3 positive integers and decides if they form an equilateral triangle, an isosceles triangle, a scalene triangle, or if they don't form a triangle at all
- The logic of the program is clear, but complex, due to the relationships between the inputs and the outputs
- For simplicity, we've removed the boundary requirement -- that each side length be between 1 and 200 -- and almost all error checking

1. Program Triangle “Determine type of triangle”

2. Integer a,b,c

3. Boolean IsATriangle

“Step 1: Get the input

4. (A) Print(“Enter 3 integers that are the sides of a triangle”)

5. (A) Read(a, b, c)

6. (A) Print(“Side a is “, a)

7. (A) Print(“Side b is “, b)

8. (A) Print(“Side c is “, c)

“Step 2: Do we have a triangle?”

9. (B) If (a < b + c) AND (b < a + c) AND (c < a + b)

10. (C) Then IsATriangle = True

11. (D) Else IsATriangle = False

12. (E) Endif

Triangle Program “Implementation”

“Step 3: Determine the type of triangle”

13. (F) If IsATriangle

14. (H) Then If (a = b) AND (b = c)

15. (I) Then Print(“Equilateral”)

16. (J) Else If (a ≠ b) AND (a ≠ c) AND (b ≠ c)

17. (K) Then Print(“Scalene”)

18. (L) Else Print(“Isosceles”)

19. (M) Endif

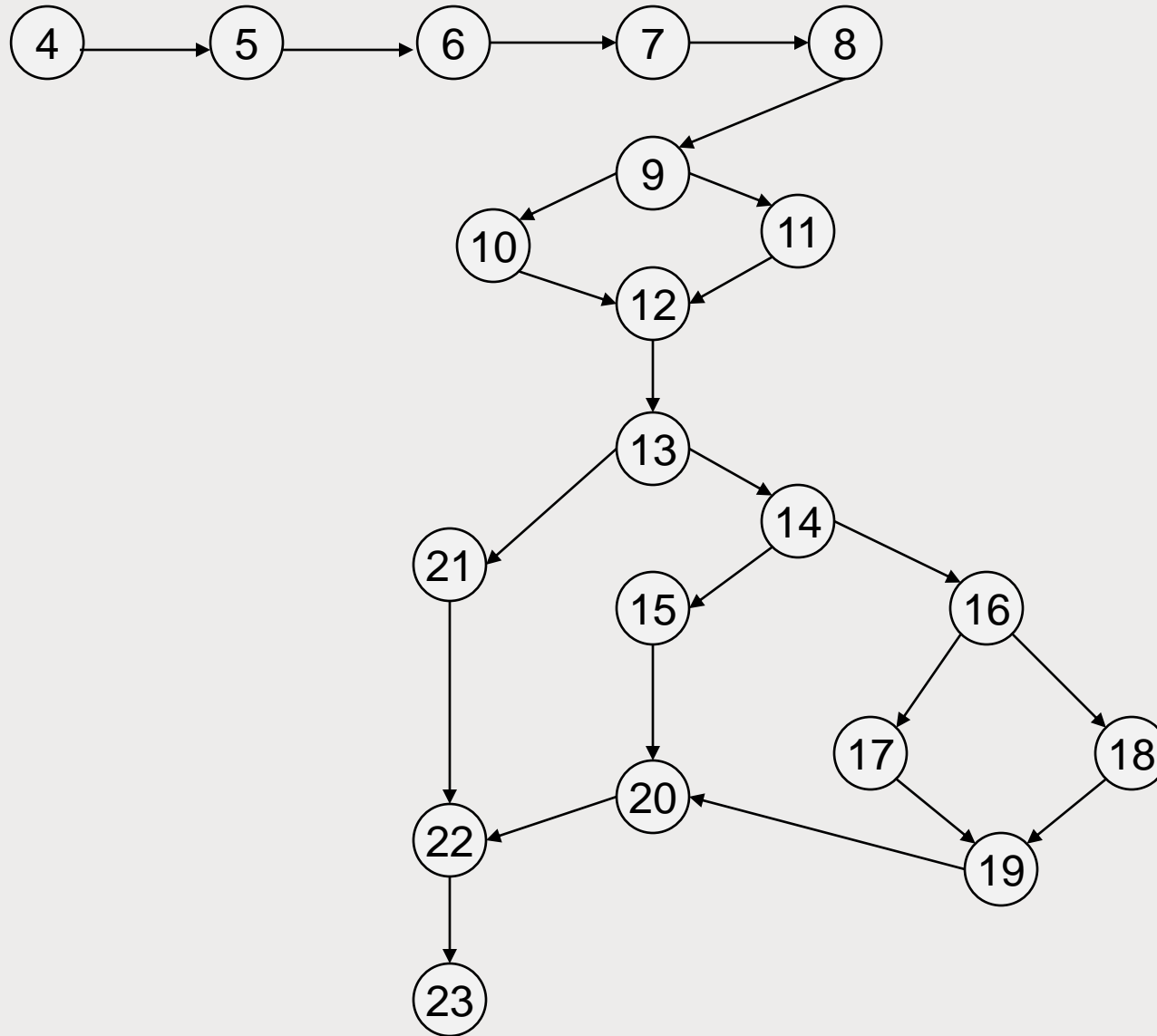
20. (N) Endif

21. (G) Else Print(“Not a triangle”)

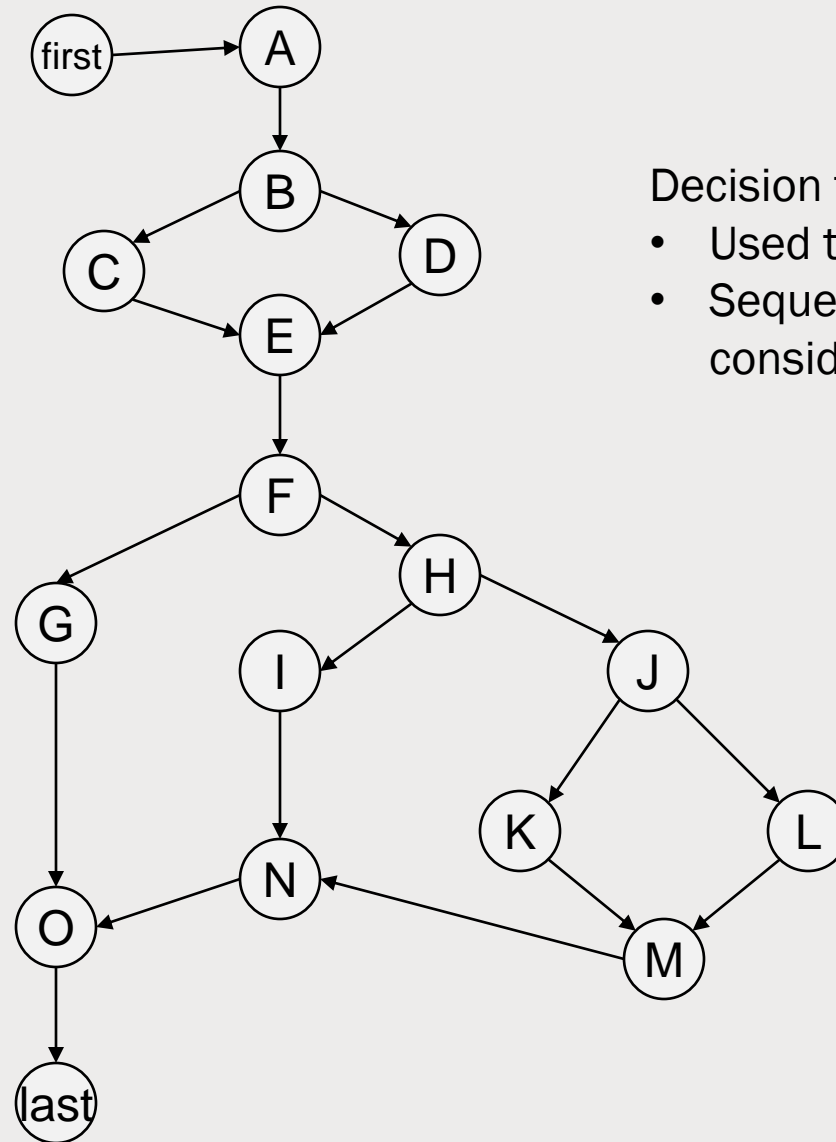
22. (O) Endif

23. {last} End Triangle

Construct the Program Graph for the program module under test



Condense the Program Graph to a DD-Path Graph



Decision to Decision paths

- Used to condense the program graph
- Sequences between decisions are considered as one node

Count and List the Syntactical Paths

There are eight possible paths through the graph

1. first – A – B – C – E – F – G – O – last
2. first – A – B – D – E – F – G – O – last
3. first – A – B – C – E – F – H – I – N – O – last
4. first – A – B – D – E – F – H – I – N – O – last
5. first – A – B – C – E – F – H – J – K – M – N – O – last
6. first – A – B – D – E – F – H – J – K – M – N – O – last
7. first – A – B – C – E – F – H – J – L – M – N – O – last
8. first – A – B – D – E – F – H – J – L – M – N – O – last

Rules for the Triangle Program

It's the discovery of these rules that uses the tester's knowledge of the program and its domain.

For our program, $(C) \Rightarrow (H)$, and $(D) \Rightarrow (G)$.

If node C is traversed, then we must traverse node H. This eliminates Path 1

If node D is traversed, then we must traverse Node G. This eliminates Paths 4, 6 and 8

- ~~1. first – A – B – C – E – F – G – O – last~~
2. first – A – B – D – E – F – G – O – last
3. first – A – B – C – E – F – H – I – N – O – last
- ~~4. first – A – B – D – E – F – H – I – N – O – last~~
5. first – A – B – C – E – F – H – J – K – M – N – O – last
- ~~6. first – A – B – D – E – F – H – J – K – M – N – O – last~~
7. first – A – B – C – E – F – H – J – L – M – N – O – last
- ~~8. first – A – B – D – E – F – H – J – L – M – N – O – last~~

Test Cases

2. $a = 3, b = 3, c = 7$ (Not a Triangle)
3. $a = 5, b = 5, c = 5$ (Equilateral)
5. $a = 3, b = 4, c = 5$ (Scalene)
7. $a = 6, b = 4, c = 6$ (Isosceles)

Summary

- The idea of checking all paths is to verify that code hasn't been implemented that's not needed
- Path analysis also verifies that the program can *correctly* execute its code with valid inputs
- Knowing you've executed all feasible paths correctly can give you confidence

A Final Note on Coverage

- Each project must choose its own minimum coverage criteria for release
- 100% coverage takes a lot of effort – effort that may be better spent on other testing activities

Predicate Testing

- The goal of predicate-based test generation is to generate tests from a predicate **p** that guarantee the detection of any error that belongs to a class of possible errors in the coding of **p**.
- This assumes the design of the predicate is correct. We want to test that the predicate got implemented correctly (and stays implemented correctly).

Fault model for predicate testing

Which faults are we targeting when testing for the correct implementation of predicates?

Suppose that the specification of a software module requires that an action be performed when the condition $(a < b) \vee (c > d) \wedge e$ is true.

Here a , b , c , and d are integer variables and e is a Boolean variable.

Some possible Boolean operator faults

Correct predicate: $(a < b) \vee (c > d) \wedge e$

$(a < b) \wedge (c > d) \wedge e$

Incorrect Boolean operator

$(a < b) \vee \neg (c > d) \wedge e$

Incorrect negation operator

$(a < b) \wedge (c > d) \vee e$

Incorrect Boolean operators

Some possible Relational operator faults

Correct predicate: $(a < b) \vee (c > d) \wedge e$

$(a == b) \vee (c > d) \wedge e$

Incorrect relational operator

$(a == b) \vee (c \leq d) \wedge e$

Two relational operator faults

Goal of predicate testing

The goal of predicate testing is to generate a *minimal* test set T such that, given a correct predicate p_c and a faulty version p_i , there is at least one test case $t \in T$ for which p_c and its faulty version p_i evaluate to *different* truth values.

Such a test set is said to guarantee the detection of any fault of the kind in the fault model introduced above (in our case, Boolean operator or Relational operator faults).

We want to catch any inadvertent changes to that predicate from later updates.

Test Cases from MC/DC constraints

Let $p_c = a < b \wedge c > d$

Constraints: The Constraint Set we get for MC/DC of the Boolean expression p_c is $\{ (t, t), (t, f), (f, t) \}$

Test Cases: Input data to meet the MC/DC constraints

t_1 : $\langle a=1, b=2, c=1, d=0 \rangle$ satisfies constraint (t, t) ,
i.e., $a < b$ is true and $c > d$ is also true.

t_2 : $\langle a=1, b=2, c=1, d=2 \rangle$ satisfies constraint (t, f)

t_3 : $\langle a=1, b=0, c=1, d=0 \rangle$ satisfies constraint (f, t)

These constraints give us a minimal constraint set for MC/DC coverage, but they would NOT catch every operator fault. [e.g. $p_i: a \leq b \wedge c > d$]

Data constraints on input values to detect operator faults

Predicates ↓ Test Input Data Constraints →	Test 1	Test 2	Test 3	Test 3	Test 5
	a1=b1, c1>d1	a2>b2 c2>d2	a3<b3, c3<d3	a4<b4, c4=d4	a5<b5. c5>d5
(a < b) AND (c > d)	F	F	F	F	T
(a = b) AND (c > d)	T	F	F	F	F
(a < b) OR (c > d)	T	T	T	T	T
(a ≤ b) AND (c > d)	T	F	F	F	T
(a ≥ b) AND (c = d)	F	F	F	F	F
(a < b) AND (c ≤ d)	F	F	T	F	F



FAULT-BASED TESTING



Fault-based testing

- X-based testing
 - $\Rightarrow X$ is a source of information for testing
- From a *model* of typical faults in a program
 - *Assess the quality of test suites*
 - *Generate useful test cases*
- Mutation-testing
 - *A fault-based testing approach*

Judging test suite effectiveness

- Seed program with faults
- Execute test suite on the program
- Compute the fraction of seeded faults that are detected by the test suite
- A good measure of effectiveness?
 - *If seeded faults are representative of the real faults ...*

Assumptions in fault-based testing

- Basic assumptions
 - *There is a “good” fault model*
 - *We know how to obtain one*
- Competent programmer hypothesis
 - *Programmers are competent*
 - The program under test is *close to being correct*
- Coupling effect
 - *Complex faults are coupled to simple faults*
 - Test data that is sensitive to simple faults will be sensitive to complex faults

Generating fault-based tests

- Seed “representative” faults in a program
- Create tests that detect the seeded faults
- Expectation
 - *The tests will detect actual faults*
- Mutation Analysis
 - *Mutants: Programs seeded with faults*
 - *Mutation operators: Patterns applied to seed faults*
 - *Mutation Testing: Creating tests that differentiate mutants from original*

Mutant

- A (small) variation of the original program
 - *Valid, if syntactically (and semantically) correct*
 - Must at least pass compilation
 - *Useful, if behaviorally different for a small subset of inputs*
 - What if behavior is obviously different?
 - *E.g. Mutant always causes null-pointer exception on execution?*

Mutation Operators

- Syntactic change applied to a program to obtain a *mutant*
 - *Language specific*
- Examples
 - *Absolute Value Insertion*
 - *Arithmetic/Relational/Logical/Assignment Operator Replacement*
 - *Scalar Variable Replacement*
 - *Unary Operator Insertion/Deletion*
 - *Forced-failure-at-statement*
 - *Fail-on-zero insertion*

Mutants: Killed and Live

- A mutant is *killed* if some test case in a test suite is able to differentiate between that mutant and the original program
- A mutant that is not killed by any test case in a test suite is *live*
 - *Under what conditions may a mutant survive?*

Considerations for testing

- A mutant may be behaviorally equivalent to the original
 - `for (int i = 0; i < 10; ++i) ...`
 - `for (int i = 0; i != 10; ++i) ...`
 - *Equivalent mutants may be eliminated, when possible*
 - *Use a threshold required for the number of mutants that must be killed by the tests*
- Tests may not differentiate between mutants and original program
 - *$(a \leq b)$ and $(a \geq b)$ cannot be differentiated if $a == b$*
 - *Ineffective tests may be discarded*

Mutation testing process

- Create mutants for a given program
- Eliminate equivalent mutants, when possible
- Generate test cases and execute those on the program as well as the mutants
- Eliminate tests that are ineffective
- Repeat until *kill-threshold* is reached
- Determine if the program *passes* all tests generated

Practical issues

- Lot of mutants generated
 - *Typically one mutation operation at one location to obtain a mutant*
- Executing test cases on mutants is expensive
 - *Running each test case to completion*
 - *Running each test case on every mutant*
- Variations of mutation analysis to address these
 - *Using an effective subset of mutation operators*
 - *Weak mutation*
 - *Statistical mutation*

Comparing effectiveness of mutation operators

- Suppose
 - *op-A is mutation operator*
 - *Test Suite A is specifically designed to kill mutants produced by op-A*
 - *Test Suite A also kills mutants produced by another op-B with high probability*
- Then *op-A is more effective than op-B*
- Research and empirical studies have established some effective collection of operators
 - *E.g., Unary-operator insertion, modification of unary and binary operators (see e.g., Amman and Offutt text)*

Weak mutation

- Observe intermediate state of mutant as well as original program when testing
- Kill the seeded fault as soon as an intermediate state following the fault differs
 - *Do not wait or expect difference to show up at program output*
 - *Once all seeded faults are killed*
- Executed using a meta-mutant which includes several seeded faults instead of one

Statistical mutation

- Create a random sample of mutants on which the test suite is to be executed
 - *Sample must be a valid statistical model of occurrence frequencies of real faults*
- Useful for assessing effectiveness of a test suite
 - *Assuming tests are not developed to kill specific mutants, this method could be used to assess how thorough the test suite is*

Fault estimation using seeding

- Testing tells how many faults have been found
- Problem: How many faults remain?
- If seeded faults are representative of actual faults, then seeded faults that are not detected after testing can be used to estimate the actual faults that still remain

Subsumption of test criteria

- Mutation can be considered to subsume a wide-range of structural coverage criteria
 - *Typical proof: For a given coverage criterion look for certain mutation operators such that killing mutants produced by those operators would satisfy the structural criterion*
 - *Examples:*
 - “Bomb statements” for statement coverage
 - Relational/Conditional/Logical operator replacement for condition coverage

Summary

- Mutation testing relies on seeded faults being a *good* model of the actual faults
- When the assumptions hold, it is useful for assessing effectiveness of test suites and also designing test suites
- Application to software is more challenging and therefore not widely adopted in practice
- Often used in software testing research to compare effectiveness of testing techniques
- Fault-based testing is widely used in semiconductor manufacturing

Next Week

- Test Management and Reporting
 - *Read Chapter 5 (Including parts of Sec 5.2 that we reviewed earlier)*
- Test Execution and Oracles
 - *Peruse: <https://ix.cs.uoregon.edu/~michal/book/slides/pdf/PezzeYoung-Ch17-execution.pdf>*