COMP5111 – Fundamentals of Software Testing and Analysis

Program-based Mutation Testing

Shing-Chi Cheung

Computer Science & Engineering
HKUST

Slides adapted from www.introsoftwaretesting.com by Paul Ammann & Jeff Offutt

Background of Mutation Testing

- Also known as **mutation analysis**
  - Proposed by Richard Liption as a student in 1971.
  - Richard DeMillo, Richard Lipton and Fred Sayward. Hints on test data selection: Help for the practicing programmer. IEEE Computer 11(4), 1978.
- Application (mutation testing)
  - Generate a pool of mutants by modifying a program slightly
  - Measure the adequacy of a test suite by counting the proportion of mutants killed (mutation coverage)
- Alternative applications (mutation analysis)
  - Use mutants to seed faults artificially, especially to estimate recall
  - Generate mutants for program repair

Using the Syntax to Generate Mutants

- Lots of software artifacts follow strict syntax rules
- Syntactic rules can come from many sources
  - Programs → Program-based
  - Integration elements → Integration
  - Design documents → Model-based
  - Input descriptions → Input-based (a major fuzzing approach)
- Mutants are created with two general goals
  - Cover the syntax in some way
  - Cover various semantic/behavioral differences

Program-based Mutation Testing

Original Method

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
        if (B < A)
        {
            minVal = B;
         }
         return (minVal);
} // end Min
```

6 mutants

Each represents a separate program

With Embedded Mutants

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
Δ 1     minVal = B;
        if (B < A)
Δ 2     if (B > A)
Δ 3     if (B < minVal)
        {
            minVal = B;
Δ 4         Bomb ();
Δ 5         minVal = A;
Δ 6         minVal = failOnZero (B);
        }
        return (minVal);
} // end Min
```

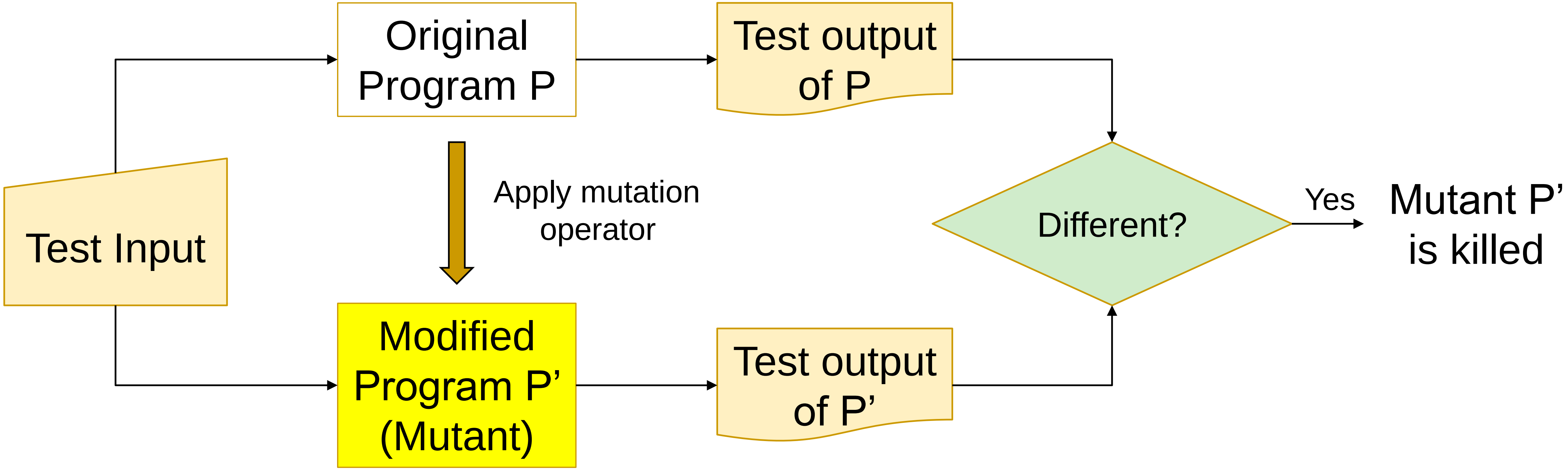
Replace one variable with another

Changes operator

Immediate runtime failure … if reached

Immediate runtime failure if B==0 else does nothing

Process of Program-based Mutation Testing



The goal is to check if the test input can detect a syntactic coding mistake in P

Killing Mutants

Given a mutant $m \in M$ for a (ground string) program P and a test t, t is said to <u>kill</u> m if and only if the output of t on P is different from the output of t on m.

- **If mutation operators are designed well, the resulting tests will be very powerful**
- Different operators must be defined for different programming languages and goals
- Testers can keep adding tests until all mutants have been killed
  - <u>Dead mutant</u> : A test case has killed it
  - <u>Stillborn mutant</u> : **Syntactically illegal**
  - <u>Trivial mutant</u> : Almost every test can kill it
  - <u>Equivalent mutant</u> : No test can kill it (equivalent to original program)

Syntax-Based Coverage Criteria

Mutation Coverage (MC) : For each $m \in M$, TR contains exactly one requirement, to kill m.

- The RIP model from Lecture 1:
  - Reachability : The test causes the faulty statement to be reached (in mutation – the mutated statement)
  - Infection : The test causes the faulty statement to result in an incorrect state
  - Propagation : The incorrect state propagates to incorrect output
- The RIP model leads to two variants of mutation coverage …

Syntax-Based Coverage Criteria

Reachability
Infection
Propagation

- Strongly Killing Mutants (satisfies RIP):
  - Given a mutant $m \in M$ for a program *P* and a test *t*, *t* is said to strongly kill *m* if and only if the output of *t* on *P* is different from the output of *t* on *m*

Reachability
Infection

- Weakly Killing Mutants (satisfies RI):
  - Given a mutant $m \in M$ that modifies a location *l* in a program *P*, and a test *t*, *t* is said to weakly kill *m* if and only if the state of the execution of *P* on *t* is different from the state of the execution of *m* immediately on *t* after *l*
  - Weakly killing satisfies reachability and infection, but not propagation

Weak Mutation

Weak Mutation Coverage (WMC) : For each $m \in M$, TR contains exactly one requirement, to weakly kill m.

- Weak mutation is so named because it is easier to kill mutants under this assumption (e.g., a function that returns void or bool)
- Weak mutation also requires less analysis
- Some mutants can be killed under weak mutation but not under strong mutation (no propagation)
- In practice, there is little difference
  - A test suite that fulfills weak mutation coverage can detect most of the faults detected by a test suite that fulfills strong weak mutation coverage

Strong vs Weak Mutation Example

- Mutant 1 in the Min( ) example is:

- The complete test specification to kill Mutant 1:
- Reachability : *true*   // Always get to that statement
- Infection : $A \neq B$
- Propagation: $(B < A) = false$ // Skip the next assignment
- Full Test Specification : $true \wedge (A \neq B) \wedge ((B < A) = false)$

  $\equiv (A \neq B) \wedge (B \geq A)$

  $\equiv (B > A)$

- ($A = 5, B = 3$) will weakly kill mutant 1, but not strongly

Original Method

```
int Min (int A, int B)
{
      int minVal;
      minVal = A;
      if (B < A)
            minVal = B;
      return (minVal);
} // end Min
```

Mutant 1

```
int Min (int A, int B)
{
      int minVal;
      minVal = B;
      if (B < A)
            minVal = B;
      return (minVal);
} // end Min
```

Equivalent Mutation Example

- Mutant 3 in the Min() example is equivalent:

- The infection condition is:
  "(B < A) != (B < minVal)"
- However, the previous statement was:
  "minVal = A"
  - Substituing, we get: "(B < A) != (B < A)"
- Thus no input can kill this mutant

Original Method

```
int Min (int A, int B)
{
      int minVal;
      minVal = A;
      if (B < A)
          minVal = B;
      return (minVal);
} // end Min
```

Mutant 3

```
int Min (int A, int B)
{
      int minVal;
      minVal = A;
      if (B < minVal)
          minVal = B;
      return (minVal);
} // end Min
```

Strong Versus Weak Mutation

```
1    boolean isEven (int X)
2    {
3        if (X < 0)
4            X = 0 - X;
∆ 4          X = 0;
5        if (float) (X/2) == ((float) X) / 2.0
6            return (true);
7        else
8            return (false);
9    }
```

Reachability : X < 0

Infection : X != 0

(X = -6) will kill mutant 4 under weak mutation

Propagation :

((float) ((0-X)/2) == ((float) 0-X) / 2.0)

!= ((float) (0/2) == ((float) 0) / 2.0)

That is, X is not even …

Thus (X = -6) does *not* kill the mutant under strong mutation

Mutation Coverage as a Stronger Coverage Criterion

```java
package demo;
public class Sample3 {
   public static void sample3(int y) {
      int sum = 0;
      for (int i = y; i < 15; i=i+1) {
         sum = sum + i;
      }
      if (sum > 4 + y)
         System.out.println("hello");
      if (sum < 2) {
         System.out.println("true");
      } else {
         System.out.println("false");
      }
     System.out.println("--");
   }
}
```

- Randoop can easily generate test cases achieving 100% line coverage.
- Are we done?

Mutation Coverage as a Stronger Coverage Criterion

Pit Test Coverage Report

Package Summary

default

Number of Classes	Line Coverage		Mutation Coverage	
4	61%	59/96	30%	22/74

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
Mono.java	33%	7/21	28%	5/18
Poly.java	74%	29/39	58%	15/26
Sample3.java	94%	17/18	0%	0/15
TestLoop.java	33%	6/18	13%	2/15

Coverage achieved by tests generated by Randoop on four classes

Mutation Coverage as a Stronger Coverage Criterion

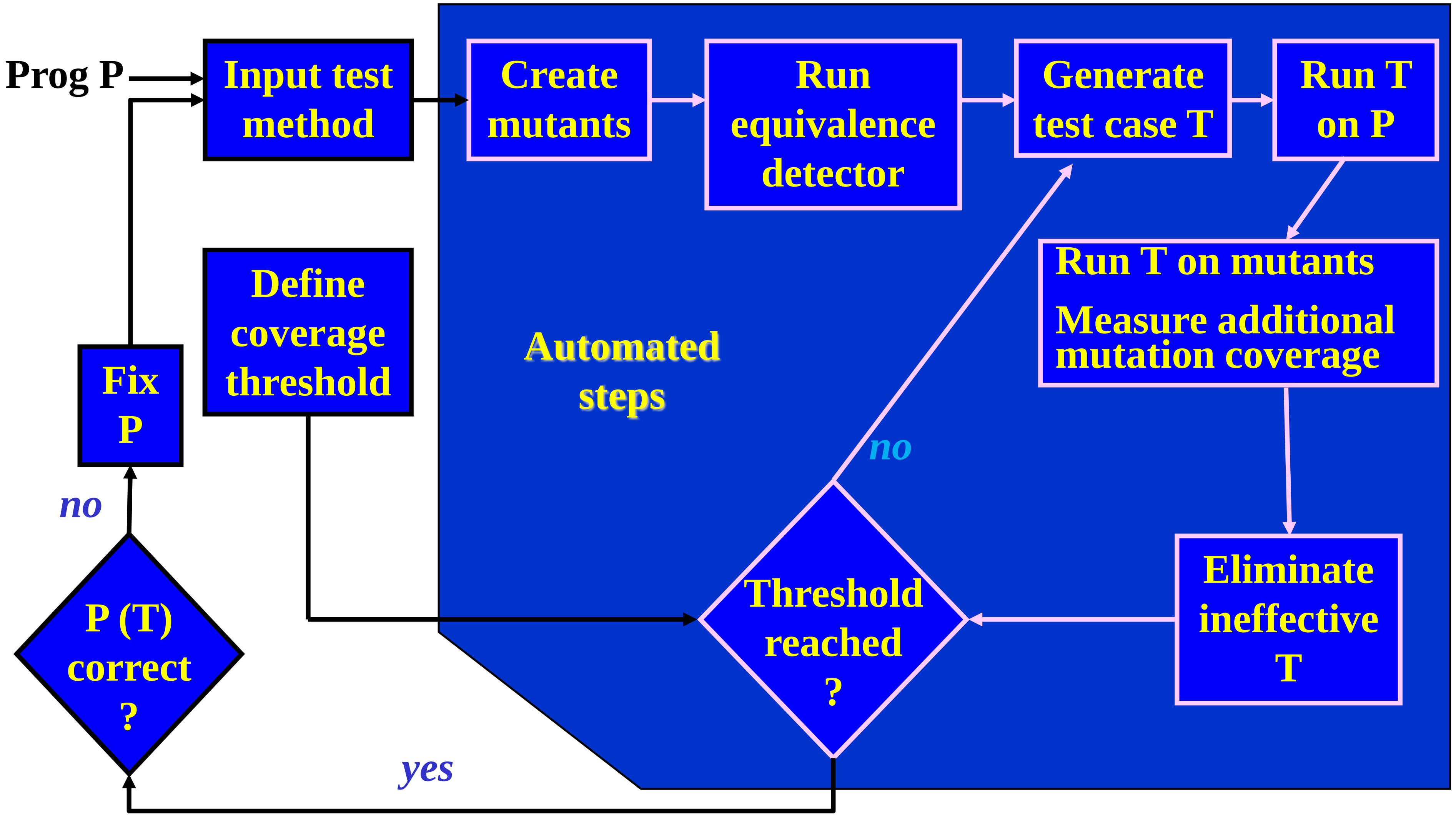
	`package demo;`
	`public class Sample3 {`
	`    public static void sample3(int y) {`
	`    int sum = 0;`
3	`    for (int i = y; i < 15; i=i+1) {`
1	`        sum = sum + i;`
	`    }`
3	`    if (sum > 4 + y)`
1	`        System.out.println("hello");`
2	`    if (sum < 2) {`
1	`        System.out.println("true");`
	`    } else {`
1	`        System.out.println("false");`
	`    }`
1	`  System.out.println("--");`
	`  }`
	`}`

Mutation Coverage by Randoop and Evosuite

	Name	Line Coverage	Mutation Coverage
Coverage achieved by Randoop	TestLoop.java	33% 7/21	12% 2/17
Coverage achieved by Evosuite	TestLoop.java	95% 20/21	24% 4/17

Mutation coverage is generally considered the finest coverage criterion in practice.

Testing Programs with Mutation



Why Mutation Works

Fundamental Premise of Mutation Testing

If the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects that fault

- This is not absolute !
- The mutants guide the tester to a very effective set of tests
- A very challenging problem :
  - Find a fault and a set of mutation-adequate tests that do not find the fault
- Of course, its effectiveness depends on the mutation operators …

Designing Mutation Operators

- At the method level, mutation operators for different programming languages are similar
- Mutation operators do one of two things:
  - Mimic typical programmer mistakes (incorrect variable name)
  - Encourage common test heuristics (cause expressions to be 0)
- Researchers design lots of operators, then experimentally *select* the most useful

Effective Mutation Operators

If tests that are created specifically to kill mutants created by a collection of mutation operators *O* = {*o1*, *o2*, …} also kill mutants created by all remaining mutation operators with very high probability, then *O* defines an effective set of mutation operators

- Refer to Appendix for 11 commonly used mutation operators

Subsumption of Other Criteria

- Mutation is widely considered the strongest test criterion
  - And most expensive !
- Mutation subsumes the following criteria by including specific mutation operators
  - Node coverage
  - Edge coverage
  - Clause coverage
  - General active clause coverage
  - Correlated active clause coverage
  - All-defs data flow coverage
- Recent studies find that strong mutation is more correlated to fault detection than weak mutation

Mutation Testing

- The number of test requirements for mutation depends on two things
  - The syntax of the artifact being mutated
  - The set of mutation operators
- Mutation testing is difficult to apply manually
- Mutation testing is very effective – considered the "golden standard" of testing
- Mutation testing is often used to evaluate the adequacy of test suites selected for other criteria
  - By mutation coverage of the test suites satisfying a criterion C

Questions About Mutation

- Should more than one operator be applied at the same time ?
  - Should a mutated string contain one mutated element or several?
  - Almost certainly not – multiple mutations can interfere with each other
  - Extensive experience with program-based mutation indicates not
- Should every possible application of a mutation operator be considered ?
  - Necessary with program-based mutation
- Mutation operators exist for several languages
  - Several programming languages (*Fortran, Lisp, Ada, C, C++, Java*)
  - Specification languages (*SMV, Z, Object-Z, algebraic specs*)
  - Modeling languages (*Statecharts, activity diagrams*)
  - Input grammars (*XML, SQL, HTML*)

How well are mutation faults coupled with real faults?

EMPIRICAL STUDY

Are mutants valid substitutes of real faults?

A study was published in FSE14

- 357 real faults (480 reproducible failing tests) in 5 open-source projects
  - Chart, Closure, Math, Time, Lang (Defect4J)
  - A total of 211K lines of code were tested
- 230,000 mutants generated for these faulty program versions
- 35,141 generated test suites detecting 198 of the real faults
  - 28,318 test suites generated by Evosuite (avg. size 68), detecting 182 faults
  - 3,387 test suites generated by Randoop (avg. size 6,929), detecting 90 faults
  - 3,436 test suites generated by JCrasher (avg. size 47,928), detecting 2 faults

Rene Just, et al.: Are Mutants a Valid Substitute for Real Faults in Software Testing? FSE 2014: 654-665.

Are mutants valid substitutes of real faults?

- Findings:
  - 263 (73%) real faults are coupled to some mutants, i.e., tests killing these mutants can expose the coupled real faults
    - Real faults are more often coupled to mutants generated by **conditional/relational operator replacement** **and** **statement deletion**
  - 32 (10%) real faults requires stronger or new mutation operators
  - 63 (17%) real faults involve algorithmic changes or code deletion. They are not coupled to any mutants even with an extended set of mutation operators
  - 258/480 failing tests kill 2 new mutants beyond those killed by passing tests written by developers

Implication?

  - 222/480 failing tests kill more than 2 new mutants (avg. 28) beyond those killed by passing tests written by developers

Rene Just, et al.: Are Mutants a Valid Substitute for Real Faults in Software Testing? FSE 2014: 654-665.

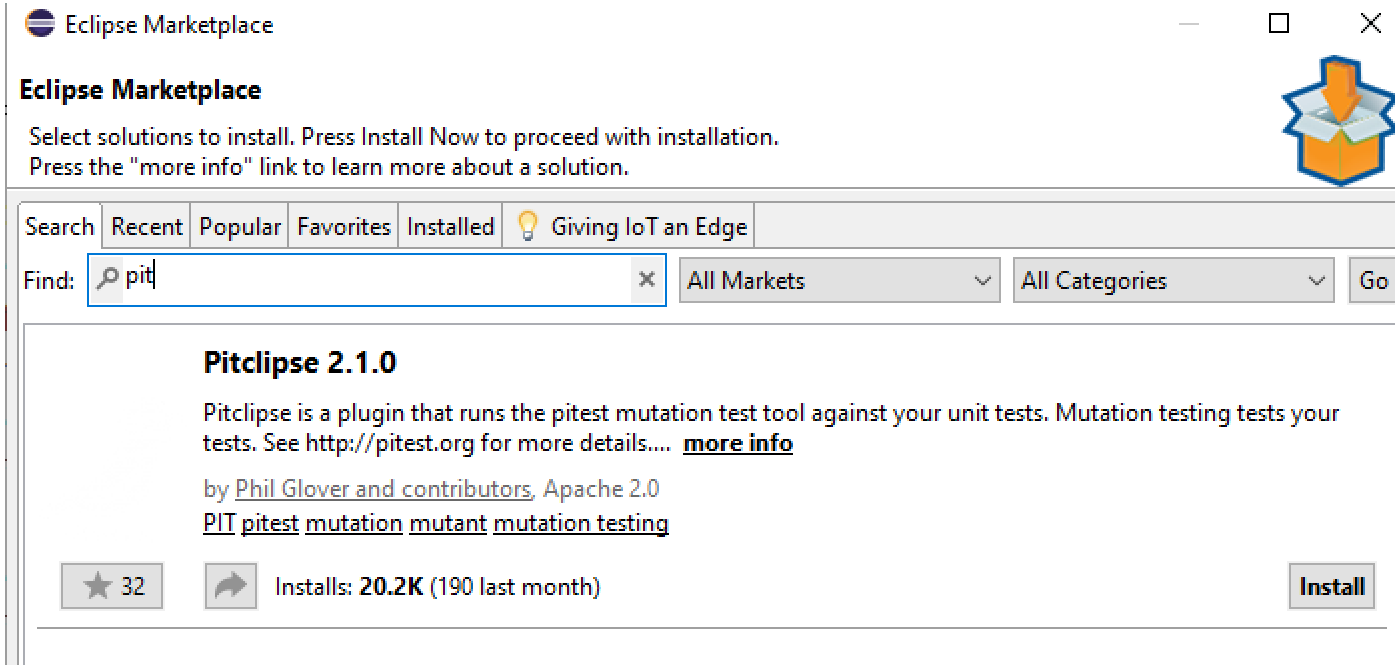
How about using mutation scores on C programs?

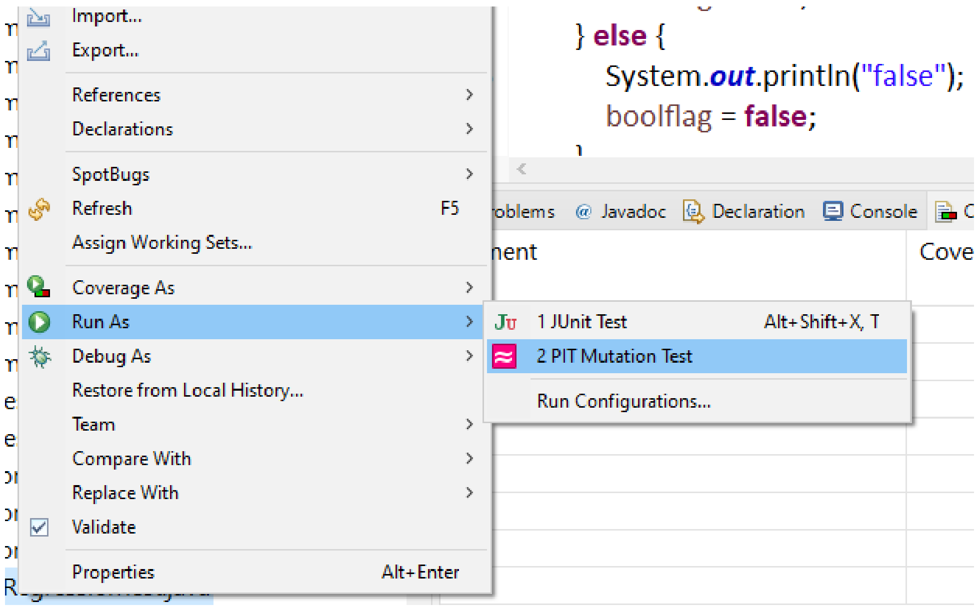
- [Chekam et al., ICSE17]
  - Mutation testing provides valuable guidance for improving test suites and revealing real faults
  - There is a strong connection between mutation score increase and fault detection at higher score levels
- [Papadakis et al., ICSE18]
  - When the mutation score is low, its correlation with fault detection is weak
  - When the mutation score is high, its correlation with fault detection is strong
- Threat to validity
  - Since mutation coverage is very fine, a higher mutation score often require a larger test suite
  - The correlation between mutation score and fault detection is partially derived from the large test suite size

Tool – PIT /Pitclipse

- Check mutation coverage of a given test suite.
- Install Pitclipse at Eclipse Market Place.
- Select the test suite and run as PIT Mutation Test.
- Click **PIT Mutations** to list all generated mutants.
- Click **PIT Summary** to find out proportion of mutants killed by the selected test suite.
- More information: https://pitest.org/
- Video:https://www.taringamberini.com/en/blog/presentation/mutation-testing-with-pit/

Pitclipse at Eclipse Marketplace





If you fail to install Pitclipse plugin directly at Eclipse Market, use the Pitclipse software update site address under installation of new software: https://pitest.github.io/pitclipse-releases/

Tool – PIT / Pitclipse

Problems | Javadoc | Declaration | Console | Bug Explorer | Randoop | Coverage | PIT Mutations | PIT Summary

Pit Test Coverage Report

Package Summary

demo

Number of Classes	Line Coverage		Mutation Coverage	
7	37%	48/129	36%	29/80

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
Circle.java	0%	0/12	0%	0/7
GeometricObject.java	0%	0/13	0%	0/4
HelloWorld.java	0%	0/13	0%	0/2
Mono.java	76%	13/17	89%	8/9
MyProgram.java	0%	0/8	0%	0/9
Poly.java	92%	35/38	70%	21/30
Rectangle.java	0%	0/28	0%	0/19

Report generated by PIT 1.0.0

Appendix

For reference only

Mutation Operators for Java

1. ABS — Absolute Value Insertion:

Each arithmetic expression (and subexpression) is modified by the functions *abs()*, *negAbs()*, and *failOnZero()*.

2. AOR — Arithmetic Operator Replacement:

Each occurrence of one of the arithmetic operators +, —,*, ⁄, and % is replaced by each of the other operators. In addition, each is replaced by the special mutation operators *leftOp*, and *rightOp*.

3. ROR — Relational Operator Replacement:

Each occurrence of one of the relational operators (<, ≤, >, ≥, =, ≠) is replaced by each of the other operators and by falseOp and trueOp.

Mutation Operators for Java (2)

4. COR — Conditional Operator Replacement:

Each occurrence of one of the logical operators (and - &&, or - || , and with no conditional evaluation - &, or with no conditional evaluation - |, not equivalent - ^) is replaced by each of the other operators; in addition, each is replaced by falseOp, trueOp, leftOp, and rightOp.

5. SOR — Shift Operator Replacement:

Each occurrence of one of the shift operators <<, >>, and >>> is replaced by each of the other operators. In addition, each is replaced by the special mutation operator leftOp.

6. LOR — Logical Operator Replacement:

Each occurrence of one of the logical operators (bitwise and - &, bitwise or - |, exclusive or - ^) is replaced by each of the other operators; in addition, each is replaced by leftOp and rightOp.

Mutation Operators for Java (3)

7. ASR — Assignment Operator Replacement:

Each occurrence of one of the assignment operators (+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=) is replaced by each of the other operators.

8. UOI — Unary Operator Insertion:

Each unary operator (arithmetic +, arithmetic -, conditional !, logical ~) is inserted in front of each expression of the correct type.

9. UOD — Unary Operator Deletion:

Each unary operator (arithmetic +, arithmetic -, conditional !, logical~) is deleted.

Mutation Operators for Java

10. SVR — Scalar Variable Replacement:

Each variable reference is replaced by every other variable of the appropriate type that is declared in the current scope.

11. BSR — Bomb Statement Replacement:

Each statement is replaced by a special Bomb() function.

Further Reading

- Ming Wen, Yepang Liu, Rongxin Wu, Xuan Xie, Shing-Chi Cheung and Zhendong Su: Exposing Library API Misuses via Mutation Analysis. In International Conference on Software Engineering, Technical Research Paper, 2019
- Xiangjuan Yao, Mark Harman, Yue Jia: A study of equivalent and stubborn mutation operators using human analysis of equivalence. ICSE 2014: 919-930
- Yue Jia, Mark Harman: An Analysis and Survey of the Development of Mutation Testing. IEEE Trans. Software Eng. 37(5): 649-678 (2011)
- David Schuler, Valentin Dallmeier, Andreas Zeller: Efficient mutation testing by checking invariant violations. ISSTA 2009: 69-80
- Rene Just, et al.: Are Mutants a Valid Substitute for Real Faults in Software Testing? FSE 2014: 654-665.