Mutation Coverage

YEGOR BUGAYENKO

Lecture #16 out of 24 80 minutes

All visual and text materials presented in this slidedeck are either originally made by the author or taken from public Internet sources, such as website. Copyright belongs to their respected authors.

Example, Part I: Code Coverage

Live Code:

```
int fibonacci(int n) {
  if (n <= 2) {
    return 1;
  }
  return fibonacci(n - 1)
    + fibonacci(n - 2);
}</pre>
```

Test Code:

```
assert fibonacci(2) == 1;

assert fibonacci(3) > 0;

C_{\text{Line}} = 7/7 = 100\%

C_{\text{Statement}} = 6/6 = 100\%

C_{\text{Branch}} = 2/2 = 100\%

C_{\text{Condition}} = 2/2 = 100\%
```

Example, Part II: Mutation Coverage

Live Code:

```
int fibonacci(int n) {
   if (n <= 2) {
     return 1;
   }
   return fibonacci(n - 1)
   + fibonacci(n - 2);
}</pre>
```

Mutant #1:

```
int fibonacci(int n) {
  if (n <= 2) {
    return 1;
  }
  return fibonacci(n - 2)
  + fibonacci(n - 2);
}</pre>
```

Mutant #2:

```
int fibonacci(int n) {
  if (n <= 2) {
    return 1;
  }
  return fibonacci(n - 1)
  * fibonacci(n - 2);
}</pre>
```

Test Code:

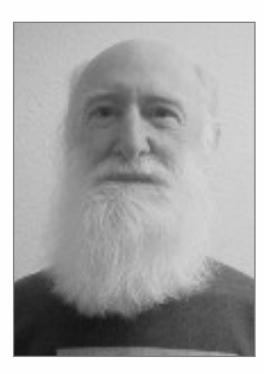
```
assert fibonacci(2) == 1;
assert fibonacci(3) > 0;
```

$C_{\mathtt{Mutants}} = 0/2 = 0\%$

Mutation Coverage

Some Mutation Operators

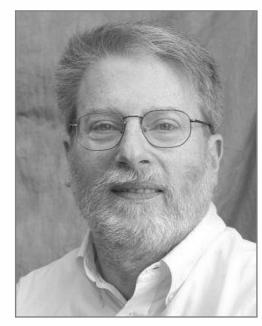
- Statement deletion
- Statement duplication or insertion
- Replacement of boolean subexpressions with TRUE and FALSE
- Replacement of some arithmetic operations, e.g. + to *, to /
- Replacement of some boolean relations, e.g. > to >=, == to <=
- Replacement of variables with others from the same scope
- Remove method body



"It is a truism that good software is easy to maintain."

Richard G. Hamlet, *Testing Programs with the Aid of a Compiler*, IEEE Transactions on Software Engineering, 4, 1977

"Dr. Hamlet presented an early testing system that was embedded in a compiler and performed a version of instrumented weak mutation. Although the method differed significantly from later mutation systems, Hamlet's system seems to be the first mutation-like testing system." — Jefferson A. Offutt and Stephen D. Lee, *How strong is weak mutation?*, Proceedings of the Symposium on Testing, Analysis, and Verification, 1991



RICHARD J. LIPTON

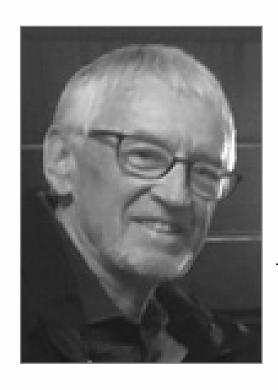
"Our groups at Yale University and the Georgia Institute of Technology have constructed a system whereby we can determine the extent to which a given set of test data has <u>adequately</u> tested a Fortran program by direct measurement of the number and kinds of errors it is capable of uncovering."

— Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4): 34–41, 1978



"A test set is <u>adequate</u> if it can distinguish the subject program from a collection of similar programs, called mutants, obtained by making <u>small</u> syntactic modifications to the subject program."

— Timothy A. Budd. Mutation analysis: Ideas, examples, problems and prospects. 1981



"In weak mutation testing method, tests are constructed which are guaranteed to force program statements which contain certain classes of errors to act incorrectly during the execution of the program over those tests."

 William E. Howden, Weak Mutation Testing and Completeness of Test Sets, IEEE Transactions on Software Engineering 4, 1982

Weak vs. Strong Mutation Testing

Live Code:

```
int fibonacci(int n) {
   if (n <= 2) {
     return 1;
   }
   return fibonacci(n - 1)
   + fibonacci(n - 2);
}</pre>
```

Mutant:

```
int fibonacci(int n) {
    if (n <= 1) {
        return 1;
     }

return fibonacci(n - 1)
     + fibonacci(n - 2);
}</pre>
```

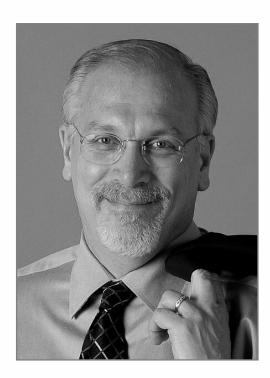
Tests Suite:

```
fibonacci(10) == 55;
fibonacci(11) == 89;
fibonacci(12) == 144;
```



"Our results indicate that weak mutation can be applied in a manner that is almost as effective as mutation testing, and with significant computational savings."

— Jeff Offutt and Stephen D. Lee, *An Empirical Evaluation* of Weak Mutation, IEEE Transactions on Software Engineering 20(5), 1994



"A mutant operator mutates <u>one syntactic entity</u> of a program. Further, only one mutant operator is applied at a time to the program under test."

— Hiralal Agrawal, <u>Richard A. DeMillo</u>, R. Hathaway, William Hsu, Wynne Hsu, Edward W. Krauser, Rhonda J. Martin, Aditya P.Mathur, Eugene Spafford, *Design of Mutant Operators for the C Programming Language*, Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, 1989

List of Mutant Operators for ANSI C

Domain	Description	Page
Constants	Constant replacement using global constants	63
		63
_ JHOUMED	constants	-
Constants	Constant for scalar replacement using global	63
	constants	
Constants	Required constant replacement	62
		63
+	arithmetic assignment mutation	49
Į.	arithmetic operator mutation	49
Ť	arithmetic assignment by bitwise assignment	50
†	arithmetic operator by bitwise operator	50
†	arithmetic assignment by plain assignment	50
†	arithmetic operator by logical operator	50
†	arithmetic operator by relational operator	50
†	arithmetic assignment by shift assignment	50
†	Arithmetic operator by shift operator	50
†	Bitwise assignment by arithmetic assignment	50
†	Bitwise operator by arithmetic assignment	50
‡	Bitwise assignment mutation	49
‡	Bitwise operator mutation	49
†	Bitwise assignment by plain assignment	50
†	0 11	50
†		52
†	9	50
†		50
†		50
Casts	1 0 1	53
†		50
†		50
†	Plain assignment by shift assignment	50
	Constants Constants Constants Constants † † † † † † † † † † † † † † † † † †	Constants Required constant replacement using global constants Constant replacement using local constants † arithmetic assignment mutation † arithmetic operator mutation † arithmetic operator by bitwise assignment † arithmetic operator by plain assignment † arithmetic operator by relational operator † arithmetic operator by shift operator † Arithmetic operator by arithmetic assignment † Bitwise operator by arithmetic assignment † Bitwise operator mutation † Bitwise operator mutation † Bitwise operator by logical operator † Bitwise operator by logical operator † Bitwise operator by logical operator † Bitwise operator by relational operator † Bitwise operator by relational operator † Bitwise operator by shift assignment † Bitwise operator by shift operator Casts Cast operator by cast operator † Plain assignment by arithmetic assignment † Plain assignment by bitwise assignment

"Each mutant operator belongs to one of the following categories: 1) statement mutations, 2) operator mutations, 3) variable mutations, and 4) constant mutations."

Source: Hiralal Agrawal et al., *Design of Mutant Operators for the C Programming Language*, Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, 1989



"Those mutants that compute precisely the same function are called <u>equivalent</u> mutants and the others are called inequivalent mutants."

— Phyllis G. Frankl, Stewart N. Weiss, and Cang Hu, *All-Uses vs Mutation Testing: An Experimental Comparison of Effectiveness?*, Journal of Systems and Software 38(3), 1997

Equivalent Mutants, Example

Live Code:

```
int fibonacci(int n) {
  if (n <= 2) {
    return 1;
  }
  return fibonacci(n - 1)
    + fibonacci(n - 2);
}</pre>
```

Inequivalent Mutant:

```
int fibonacci(int n) {
  if (n <= 2) {
    return 1;
  }
  return fibonacci(n + 1)
  + fibonacci(n - 2);
}</pre>
```

Equivalent Mutant:

```
int fibonacci(int n) {
  if (n <= 2) {
    return 1;
  }
  return fibonacci(n - 2)
  + fibonacci(n - 1);
}</pre>
```

Tests:

```
fibonacci(2) == 1;
fibonacci(14) == 377;
```

↑ You can't kill this one!

Mutation Coverage

subject	LOC	mutants	duas	inequiv mutants	exec duas	failure rate
determinant	60	4489	298	4123	103	0.0008
find1	33	932	114	836	93	0.066
find2	33	932	114	859	93	0.018
matinv1	60	4303	298	3971	106	0.012
matinv2	28	1267	81	1145	62	0.014
strmatch1	22	398	49	356	49	0.032
strmatch2	23	402	56	361	54	0.062
textformat.0	26	976	50	905	42	0.066
textformat.r	26	976	50	976	42	0.066
transpose	78	5358	97	4595	88	0.023

Source: Phyllis G. Frankl, Stewart N. Weiss, and Cang Hu, *All-Uses vs Mutation Testing: An Experimental Comparison of Effectiveness?*, Journal of Systems and Software 38(3), 1997

"Mutation coverage is more effective than dua coverage for five subjects, dua coverage — for two others, and there is no significant difference for the remaining two.

A definition-use association (\underline{dua} is a triple d, u, v, such that d is a node in the program's flow graph in which variable v is defined, u is a node or edge in which v is used, and there is a definition-clear path with respect to v from d to u."



"Our analysis suggests that mutants, when using carefully selected mutation operators and after removing equivalent mutants, can provide a good indication of the fault detection ability of a test suite."

— James H. Andrews, <u>Lionel C. Briand</u> and Yvan Labiche, Is Mutation an Appropriate Tool for Testing Experiments?, Proceedings of the 27th International Conference on Software Engineering (ICSE), 2005

Table 3. Matched Pairs t-test Results – test suite size = 100

	Matched Pairs Results			
Subject Programs	Mean Af(S) – Am(S)	t-ratio	<i>p</i> -value	
Space	0.014	16.87	< 0.0001	
Replace	-0.266	-233.96	0.0000	
Printtokens	-0.344	-158.2	0.0000	
Printtokens2	-0.061	-59.39	0.0000	
Schedule	-0.298	-161.33	0.0000	
Schedule2	-0.327	-152.19	0.0000	
Tcas	-0.1128	-57.56	0.0000	
Totinfo	-0.1037	-145.78	0.0000	

"Average differences range from 6% to 34%, with an average of 22%.

If one has used mutants to assess a test technique, it will likely look more effective at detecting faults than if one has used the <u>seeded</u> faults."

Source: James H. Andrews, Lionel C. Briand and Yvan Labiche, *Is Mutation an Appropriate Tool for Testing Experiments?*, Proceedings of the 27th International Conference on Software Engineering (ICSE), 2005



"Comparing with previous mutation systems for procedural programs, <u>MuJava</u> is very fast. However, it is relatively slow when it generates and runs lots of mutants."

Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon, *MuJava:* A Mutation System for Java, Proceedings of the 28th
 International Conference on Software Engineering (ICSE),
 2006

Operator	Description		
IHD	Hiding variable deletion		
IHI	Hiding variable insertion		
IOD	Overriding method deletion		
IOP	Overridden method calling position change		
IOR	Overridden method rename		
ISI	super keyword insertion		
ISD	super keyword deletion		
IPC	Explicit call of a parent's constructor deletion		
PNC	new method call with child class type		
PMD	Instance variable declaration with parent class type		
PPD	Parameter variable declaration with child class type		
PCI	Type cast operator insertion		
PCC	Cast type change		
PCD	Type cast operator insertion		
PRV	Reference assignment with other compatible type		
OMR	Overloading method contents change		
OMD	Overloading method deletion		
OAC	Argument order change		
JTI	this keyword insertion		
JTD	this keyword deletion		
JSI	static modifier insertion		
JSD	static modifier deletion		
JID	Member variable initialization deletion		
JDC	Java-supported default constructor create		
EOA	Reference and content assignment replacement		
EOC	Reference and content assignment replacement		
EAM	Accessor method change		
EMM	Modifier method change		

Table 2: Class-level Mutation Operators for Java

"Method-level mutation operators handle primitive features of programming languages. They modify expressions by replacing, deleting, and inserting primitive operators. Class-level mutation operators handle object-oriented specific features such as inheritance, polymorphism and dynamic binding."

Source: Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon, *MuJava: A Mutation System for Java*, Proceedings of the 28th International Conference on Software Engineering (ICSE), 2006



"Three conditions must be present for a failure to be observed: 1) The location in the program that contains the fault must be reached (Reachability).

2) After executing the location, the state of the program must be incorrect (Infection). 3) The infected state must propagate to cause some output of the program to be incorrect (Propagation)."

 Paul Ammann and Jeff Offutt, Introduction to Software Testing, 2008



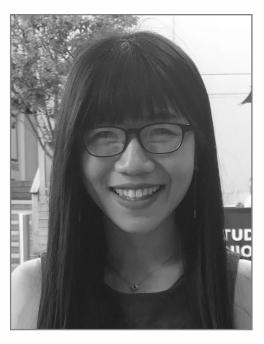
"Traditional mutation testing considers only first order mutants, created by the injection of a single fault. Often these first order mutants denote trivial faults that are easily killed. Higher order mutants are created by the insertion of two or more faults."

Yue Jia and Mark Harman, Higher Order Mutation
 Testing, Information and Software Technology 51(10), 2009



"One <u>problem</u> that prevents mutation testing from becoming a practical testing technique is the <u>high</u> computational cost of executing the enormous number of mutants against a test set."

 Yue Jia and Mark Harman, An Analysis and Survey of the Development of Mutation Testing, IEEE Transactions on Software Engineering 37(5), 2010



"PMT applies ML to build a predictive model by collecting a series of easy-to-access features (e.g., coverage and mutation operator) on already executed mutants of earlier versions of the project. Based on this model, PMT <u>predicts</u> the mutation testing results (i.e., whether each mutant is killed or not) of a new version of project without executing its mutants at all."

<u>Jie Zhang</u>, Ziyi Wang, Lingming Zhang, Dan Hao, Lei Zang, Shiyang Cheng, and Lu Zhang, *Predictive Mutation Testing*, Proceedings of the 25th International Symposium on Software Testing and Analysis, 2016

Mutation Coverage can be calculated by a few tools:

- PIT for Java
- StrykerJS for JavaScript
- Mutate++ for C++
- mutatest for Python
- mutant for Ruby

References

Timothy A. Budd. Mutation analysis: Ideas, examples, problems and prospects. 1981.

Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, 1978.