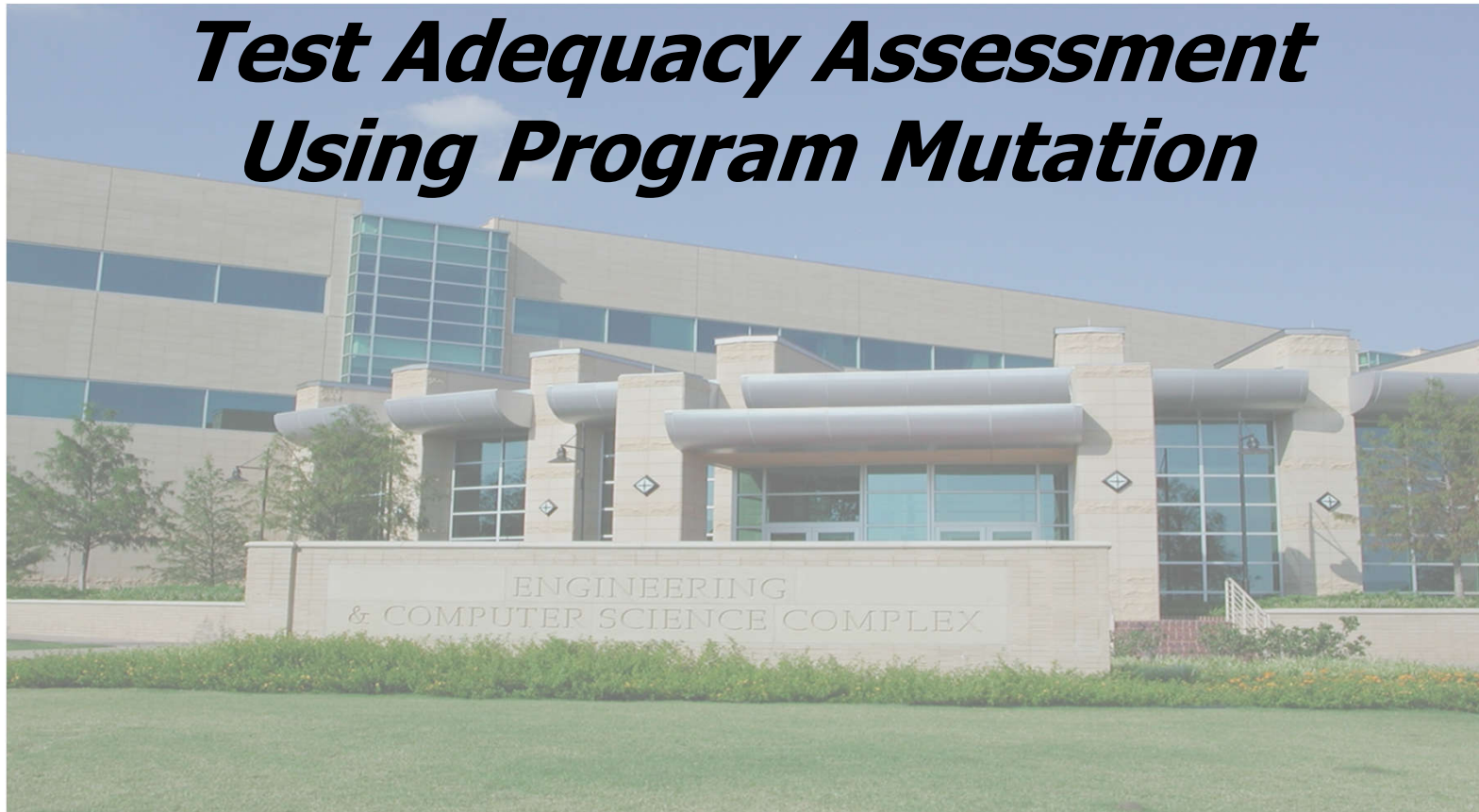


Test Adequacy Assessment Using Program Mutation



Dr. Mark C. Paulk


SE 4367 – Software Testing, Verification, Validation, and Quality Assurance

Mutation Topics

Part III. Test Adequacy Assessment and Enhancement

7. Test Adequacy Assessment Using Control Flow and Data Flow

8. Test Adequacy Assessment Using Program Mutation

- 
- Introduction
 - Mutation and Mutants
 - Test Assessment Using Mutation
 - Mutation Operators
 - Design of Mutation Operators
 - Founding Principles of Mutation Testing
 - Equivalent Mutants
 - Fault Detection Using Mutation
 - Types of Mutants
 - Mutation Operators for C
 - Mutation Operators for Java
 - Comparison of Mutation Operators
 - Mutation Testing Within Budget

What Is Program Mutation?

Suppose that program P has been tested against a test set T.

- **P has not failed on any test case in T**

Now suppose that we change P to P'.

- **mutation is the act of changing a program**

What behavior do you expect from P' against tests in T?

Caution: mutation is a significantly different way of assessing test adequacy...

Mutants

P' is known as a mutant of P.

There might be a test t in T such that $P(t) \neq P'(t)$.

- **t distinguishes P' from P**
- **t has killed P'**

There might be not be any test t in T such that $P(t) \neq P'(t)$.

- **T is unable to distinguish P and P'**
- **P' is considered live in the test process**

Mutant Equivalent

If there does not exist any test case t in the input domain of P that distinguishes P from P' then P' is said to be equivalent to P .

If P' is not equivalent to P , but no test in T is able to distinguish it from P , then T is considered inadequate.

A non-equivalent and live mutant offers the tester an opportunity to generate a new test case and hence enhance T .

Mathur, Example 8.1

Program P8.1

```
1  begin
2      int x, y;
3      input (x, y);
4      if (x < y)
5          output (x + y);
6      else
7          output (x * y);
8  end
```

Mutant M₁ of Program P8.1

```
1  begin
2      int x, y;
3      input (x, y);
4      if (x ≤ y) ← Mutation
5          output (x + y);
6      else
7          output (x * y);
8  end
```

Mutate by introducing only slight changes.

Program P8.1

```
1  begin
2      int x, y;
3      input (x, y);
4      if (x < y)
5          output (x + y);
6      else
7          output (x * y);
8  end
```

Mutant M₂ of Program P8.1

```
1  begin
2      int x, y;
3      input (x, y);
4      if (x < y)
5          output (x + y);
6      else
7          output (x / y); ← Mutation
8  end
```

Only one change was made to create mutants M₁ and M₂.

First-Order Mutants

Mutants generated by introducing only a single change are known as first-order mutants.

Second-order mutants are created by making two simple changes and so on.

First-order mutants are the ones generally used in practice.

Syntax vs Semantics

Mutate by making simple syntactic changes.

Syntax is the carrier of semantics in programming languages.

A simple syntactic change could have a drastic effect or no effect on program semantics.

Strong and Weak Mutations

Strong mutation testing uses external observations of return value and any side effects (global variables, data files).

Weak mutation testing uses internal observations of internal program state.

Two programs may be equivalent under strong mutation testing but distinguishable under weak mutation testing.

Mathur, Example 8.7

P8.1 computes $f(x,y)$:

Program P8.1

```
1  begin
2    int x, y;
3    input (x, y);
4    if (x < y)
5        output (x + y);
6    else
7        output (x * y);
8  end
```

$$f(x,y) = \begin{cases} x + y & \text{if } x < y \\ x * y & \text{otherwise} \end{cases}$$

$$T_P = \{t_1: \langle x=0, y=0 \rangle, \\ t_2: \langle x=0, y=1 \rangle, \\ t_3: \langle x=1, y=0 \rangle, \\ t_4: \langle x=-1, y=-2 \rangle\}$$

<u>t</u>	<u>f(x,y)</u>	<u>P(t)</u>
t₁	0	0
t₂	1	1
t₃	0	0
t₄	2	2

Mathur, Example 8.8

Program P8.1	Mutant ID	Mutants
1 begin		None
2 int x, y;		None
3 input (x, y);		None
4 if (x < y)	M ₁	if (x+1<y)
	M ₂	if (x<y+1)
5 then		None
6 output (x + y);	M ₃	output(x+1+y);
	M ₄	output(x+y+1);
	M ₅	output(x-y);
7 else		None
8 output (x * y);	M ₆	output((x+1)*y);
	M ₇	output(x*(y+1));
	M ₈	output(x/y);
9 end		None

Have not mutated declaration, input, then, else statements (or begin/end)

Eight live mutants (not yet distinguished from original P8.1)

$L = \{M_1, M_2, M_3, M_4, M_5, M_6, M_7, M_8\}$

Mathur, Examples 8.9-11

8.9 Select M_1 from L.

- $L = \{M_2, M_3, M_4, M_5, M_6, M_7, M_8\}$

8.10 Select t_1 : $\langle x=0, y=0 \rangle$ from T.

8.11 Execute M_1 against t_1

- $M_1(t_1) = 0, P(t_1) = 0$
- $P(t_1) = M_1(t_1)$ therefore t_1 is unable to distinguish M_1 from P

Execute M_1 against t_2

- $M_1(t_2) = 0, P(t_2) = 1$
- $P(t_2) \neq M_1(t_2)$, so M_1 is a distinguished (or killed) mutant

Mathur, Example 8.12

Program	t_1	t_2	t_3	t_4	D
P(t)	0	1	0	2	{}

Mutant					
$M_1(t)$	0	0*	NE	NE	{ M_1 }
$M_2(t)$	0	1	0	2	{ M_1 }
$M_3(t)$	0	2*	NE	NE	{ M_1, M_3 }
$M_4(t)$	0	2*	NE	NE	{ M_1, M_3, M_4 }
$M_5(t)$	0	-1*	NE	NE	{ M_1, M_3, M_4, M_5 }
$M_6(t)$	0	1	0	0*	{ M_1, M_3, M_4, M_5, M_6 }
$M_7(t)$	0	1	1*	NE	{ $M_1, M_3, M_4, M_5, M_6, M_7$ }
$M_8(t)$	U*	NE	NE	NE	{ $M_1, M_3, M_4, M_5, M_6, M_7, M_8$ }

* – distinguishing test case

NE – mutant was not executed against test case

U – undefined output

Mathur, Example 8.14

M_2 is live – is M_2 equivalent to P ?

$$f(x,y) = \begin{cases} x + y & \text{if } x < y \\ x * y & \text{otherwise} \end{cases}$$

$$g_{M_2} P(x,y) = \begin{cases} x + y & \text{if } x < y + 1 \\ x * y & \text{otherwise} \end{cases}$$

Can we find $f_P(x_1, y_1) \neq g_{M_2}(x_1, y_1)$?

- **$C_1: (x_1 < y_1) \neq (x_1 < y_1 + 1)$**
- **$C_2: x_1 * y_1 \neq x_1 + y_1$**
- **$x_1 = y_1 \neq 0$ (t_1 satisfies C_1 but not C_2)**
- **$t: \langle x=1, y=1 \rangle$ satisfies both C_1 and C_2**
- **M_2 can be distinguished by at least one test case**
 - **$P(t) = 2, M_2(t) = 1$**

Mathur, Example 8.15

At the end of Step 11, we are left with

- one live mutant, $|L| = 1$**
- seven distinguished (killed) mutants, $|D| = 7$**
- no equivalent mutant, $|E| = 0$**

$$\mathbf{MS(T) = 7 / (1 + 7) = 0.875}$$

If we enhance T with the test case t found in Example 8.14, $MS(T') = 1$.

Error Detection Using Mutation

As with any test enhancement technique, there is no guarantee that tests derived to distinguish live mutants will reveal a yet undiscovered error in P.

Empirical studies have found mutation testing to be the most powerful of all formal test enhancement techniques.

Mutation Example

Consider the following function foo that is required to return the sum of two integers x and y.

```
int foo (int x, y)
{
    return (x - y);
}
```

Clearly foo is incorrect.

Suppose that foo has been tested using a test set T that contains two tests:

$$T = \{t_1: \langle x=1, y=0 \rangle, \\ t_2: \langle x=-1, y=0 \rangle\}$$

foo returns the expected value for each test case in T.

T is adequate with respect to all control and data flow based test adequacy criteria.

Evaluate the adequacy of T using mutation.

Suppose that the following three mutants are generated from foo.

M₁
int foo (int x, y)
{
 return (x + y);
}

M₂
int foo (int x, y)
{
 return (x – 0);
}

M₃
int foo (int x, y)
{
 return (0 – y);
}

M₁ is obtained by replacing the – by a + operator

M₂ by replacing y by 0

M₃ by replacing x by 0

Execute each mutant against tests in T until the mutant is distinguished or we have exhausted all tests.

**$T = \{t_1: \langle x=1, y=0 \rangle,$
 $t_2: \langle x=-1, y=0 \rangle\}$**

Test (t)	foo(t)	$M_1(t)$	$M_2(t)$	$M_3(t)$
t_1	1	1	1	0
t_2	-1	-1	-1	0
		Live	Live	Killed

After executing all three mutants we find that two are live and one is distinguished.

Computation of mutation score requires us to determine if any of the live mutants is equivalent.

In class exercise: Determine whether or not the two live mutants are equivalent to foo and compute the mutation score of T.

Examine the two live mutants.

M_1	M_2
int foo (int x, y)	int foo (int x, y)
{	{
return (x + y);	return (x - 0);
}	}

A test that distinguishes M_1 from foo must satisfy the following condition:

$x - y \neq x + y$ implies $y \neq -y$ implies $y \neq 0$

t_3 : $\langle x=1, y=1 \rangle$

Executing foo on t_3 gives us foo(t_3) = 0

- according to the requirements, foo(t_3) = 2
- t_3 distinguishes M_1 from foo
- t_3 reveals the error

M_1	M_2
<pre>int foo (int x, y) { return (x + y); }</pre>	<pre>int foo (int x, y) { return (x - 0); }</pre>

In class exercise:

- ***Will any test that distinguishes M_1 also reveal the error?***
- ***Will any test that distinguishes M_2 reveal the error?***

Examine M_2

M_1	M_2
int foo (int x, y)	int foo (int x, y)
{	{
return (x + y);	return (x - 0);
}	}

A test that distinguishes M_2 from foo must satisfy the following condition:

$$x - y \neq x - 0 \text{ implies } y \neq 0$$

t_3 : $\langle x=1, y=1 \rangle$ distinguishes M_2 also

Guaranteed Error Detection

Sometimes there exists a mutant P' of program P such that any test t that distinguishes P' from P also causes P to fail.

Let P' be a mutant of P and t a test in the input domain of P . P' is an error-revealing mutant if the following condition holds for any t .

- $P'(t) \neq P(t)$
- $P(t) \neq R(t)$
- where $R(t)$ is the expected response of P based on its requirements

Is M_1 in the previous example an error-revealing mutant? What about M_2 ?

Distinguishing a Mutant

A test case t that distinguishes a mutant M from its parent program P must satisfy the following three conditions.

C_1 : Reachability: It must force M to follow a path from the start statement of M to the mutated statement in M .

C_2 : State Infection: If S_{in} is the state of M upon arrival at the mutant statement, and S_{out} the state after the execution of the mutated statement, then $S_{in} \neq S_{out}$

C_3 : State Propagation: The difference between S_{in} and S_{out} must propagate to the output of M such that the output of M is different from that of P .

Equivalent Mutants

The problem of deciding whether or not a mutant is equivalent to its parent program is undecidable.

- there is no way to fully automate the detection of equivalent mutants**

Empirical studies have shown that one can expect about 5% of the generated mutants to be equivalent to the parent program.

Identifying equivalent mutants is generally a manual and often time consuming – as well as frustrating – process.

A Misconception

There is a widespread misconception that any “coverage” based technique, including mutation, will not be able to detect errors due to a missing path. Consider the following.

Program under test

```
int foo (int x, y)
{
    int p = 0;
    if (x < y)
        p = p + 1;
    return (x + p * y);
}
```

Correct program

```
int foo (int x, y)
{
    int p = 0;
    if (x < y)
        p = p + 1;
    else
        p = p - 1;
    return (x + p * y);
}
```

Suggest at least one mutant M of foo that is guaranteed to reveal the error.

- M is an error-revealing mutant

Suppose T is decision adequate for foo.

- Is T guaranteed to reveal the error?

Suppose T is def-use adequate for foo.

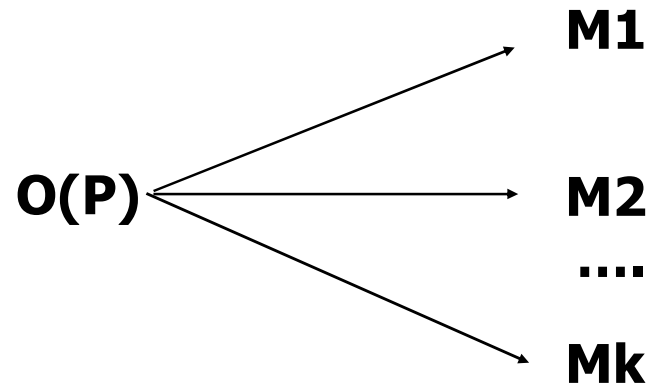
- Is T guaranteed to reveal the error?

Mutation Operators

A mutation operator O is a function that maps the program under test to a set of $k \geq 0$ mutants of P .

- aka mutagenic operator
- aka mutant operator
- aka operator

Each mutation operator is assigned a unique name.



Mathur, Example 8.18

CCR mutation operator

- replaces each occurrence of a constant c by some other constant d
- both c and d must appear in P

ABS mutation operator

- replaces each occurrence of an arithmetic expression e by the expression `abs(e)`

Basis for Mutation Operators

A mutation operator models a simple mistake that could be made by a programmer.

Several error studies have revealed that programmers – both novices and experts – make simple mistakes.

- instead of using $x < y + 1$ one might use $x < y$

Programmers make “complex mistakes” too

- the “coupling effect” explains why only simple mistakes are modeled

Language-Specific Mutation Operators

Some mutation tools provide mutation operators to enforce code and domain coverages.

- **STRP for C and Fortran replaces statements with a trap condition**
- **VDTR for C ensures negative, zero, positive values for integer variables**

Mutants must be syntactically correct.

The domain of a mutation operator is determined by the syntax rules of the programming language.

Peculiarities of language syntax may affect the kinds of mistakes that programmer make.

Generic Categories of Mutation Operators

Constant mutations

- **incorrect constant**

Operator mutations

- **incorrect operator**

Statement mutations

- **incorrectly placed statement**

Variable mutations

- **incorrectly used variable**

Examples of Mutation Operators

Mutant operator	In P	In mutant
Variable replacement	$z = x * y + 1;$	$x = x * y + 1;$ $z = x * x + 1;$
Relational operator replacement	if ($x < y$)	if($x > y$) if($x \leq y$)
Off-by-1	$z = x * y + 1;$	$z = x * (y + 1) + 1;$ $z = (x + 1) * y + 1;$
Replacement by 0	$z = x * y + 1;$	$z = 0 * y + 1;$ $z = 0;$
Arithmetic operator replacement	$z = x * y + 1;$	$z = x * y - 1;$ $z = x + y - 1;$

Goodness Criteria for Mutation Operators

The design of mutation operators is based on guidelines and experience.

- **two groups might arrive at a different set of mutation operators for the same programming language**
- **how should we judge whether a set of mutation operators is “good enough?”**

Let S_1 and S_2 denote two sets of mutation operators for language L .

- **S_1 is superior to S_2 if mutants generated using S_1 guarantee a larger number of errors detected over a set of erroneous programs.**

Constrained (or Selective) Mutation

Generally one uses a small set of highly effective mutation operators rather than the complete set of operators.

Experiments have revealed relatively small sets of mutation operators for languages such as Fortran, C, Ada, Lisp, and Java. Guidelines are based on

- error studies**
- experience with mutation**
- experiments to assess the effectiveness of mutation operators in detecting complex errors**

Competent Programmer Hypothesis

CPH states that given a problem statement, a programmer writes a program P that is in the general neighborhood of the set of correct programs.

An extreme interpretation of CPH is that when asked to write a program to find the account balance, given an account number, a programmer is unlikely to write a program that deposits money into an account.

- while such a situation is unlikely to arise, a devious programmer might certainly write such a program...**

CPH and Mutation

A more reasonable interpretation of the CPH is that the program written to satisfy a set of requirements will be a few mutants away from a correct program.

The CPH assumes that the programmer knows of an algorithm to solve the problem at hand, and if not, will find one prior to writing the program.

Mistakes will lead to a program that can be corrected by applying one or more first-order mutations.

Coupling Effect

The coupling effect has been paraphrased by DeMillo, Lipton, and Sayward as

- ***“Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors.”***

Stated alternately, again in the words of DeMillo, Lipton and Sayward, ***“... seemingly simple tests can be quite sensitive via the coupling effect.”***

Finding Complex Faults

For some input, a non-equivalent mutant forces a slight perturbation in the state space of the program under test.

This perturbation takes place at the point of mutation and has the potential of infecting the entire state of the program.

It is during an analysis of the behavior of the mutant in relation to that of its parent that one discovers complex faults.

Equivalent Mutants

Given a mutant M of program P , M is equivalent to P if $P(t) = M(t)$ for all possible test inputs t .

In strong mutation, behavior of P and M are compared at the end of execution.

In weak mutation, states of P and M are compared at some intermediate point.

Determining whether a mutant is equivalent to its parent is undecidable (equivalent to halting problem).

Implications of Mutation

A test set for program P, inadequate with respect to a set of mutants, offers an opportunity to construct new tests that may exercise P in new ways.

May reveal faults not necessarily modeled directly by the mutation operators.

Notation

P denotes a program.

P_c denotes a correct version of that program.

S is the specification for P_c .

D is the input domain of P_c derived from S.

M is a mutant of P.

Error-Revealing Mutants

M is said to be of type error-revealing for P if and only if for every t in D such that $P(t) \neq M(t)$, $P(t) \neq P_c(t)$, and that there exists at least one such test case.

t is considered to be an error-revealing test case.

Error-Hinting and Reliability-Indicating

M is said to be of type error-hinting if and only if P is equivalent to M and P_c is not equivalent to M.

M is said to be of type reliability-indicating if and only if $P(t) \neq M(t)$ for some t in D and $P(t) = P_c(t)$.

What Entities Are Not Mutated in C?

Declarations

Address operator (&)

Format strings in I/O functions

Function declaration headers

Control line

Function name indicating a function call

Preprocessor conditionals

Types of Java Mutation Operators

Java has two generic categories of mutation operators.

- **traditional mutation operators**
- **class mutation operators**
 - inheritance
 - polymorphism and dynamic binding
 - method overloading
 - OO
 - Java-specific

Java-Specific Mutation Operators

Mutop	Domain	Description
JTD	<i>this</i>	Delete <i>this</i> keyword
JSC	Class variables	Change a class variable to an instance variable
JID	Member variables	Remove the initialization of a member variable
JDC	Constructors	Remove user-defined constructors
EOA	Object references	Replace reference to an object by its contents using <i>clone()</i>
EOC	Comparison expressions	Replace <code>==</code> by <i>equals</i>
EAM	Calls to accessor methods	Replace call to an accessor method by a call to another compatible accessor method
EMM	Calls to modifier methods	Replace call to a modifier method by a call to another compatible modifier method

Comparing Languages

C has 77 mutation operators

Fortran 77 has 22 mutation operators

- **one of the earliest languages for which mutation operators were designed**
- **often referred to as traditional mutation operators**

Java has 29 mutation operators

Tools for Mutation Testing

As with any other type of test adequacy assessment, mutation based assessment must be done with the help of a tool.

There are few mutation testing tools available freely.

- **Proteum for C from Professor Maldonado**
- **muJava for Java from Professor Jeff Offutt**

Mutation and System Testing

Adequacy assessment using mutation is often recommended only for relatively small units

- **e.g., a class in Java or a small collection of functions in C**

However, given a good tool, one can use mutation to assess adequacy of system tests.

Summary – Things to Remember

Mutants, equivalent mutants

Distinguishing / killing mutants

Strong vs weak mutations

Distinguishing a mutant

- reachability, state infection, state propagation

Competent programmer hypothesis

Coupling effect

Traditional mutation operators (Fortran: 22)

Questions and Answers

