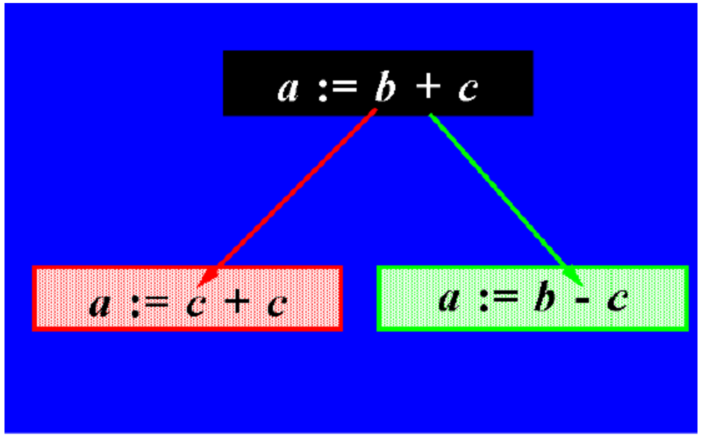
Software Verification and Validation

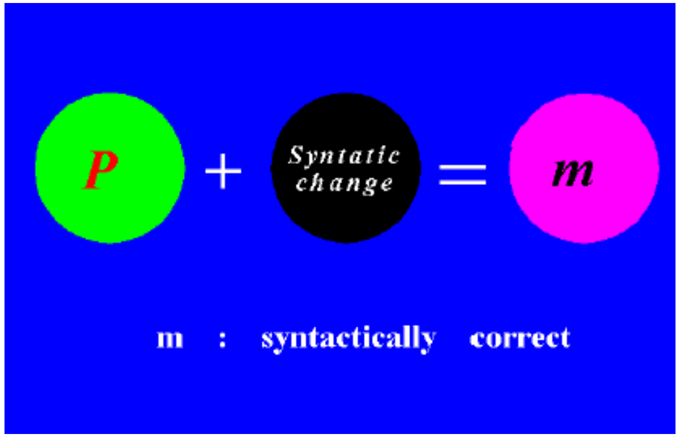
Prof. Lionel Briand
Ph.D., IEEE Fellow

Mutation Testing

Definitions

- *Fault-based Testing*: directed towards "typical" faults that could occur in a program
- Basic idea:
  - Take a program and test data generated for that program
  - Create a number of *similar* programs (mutants), each differing from the original in one small way, i.e., each possessing a fault
    - e.g., replace addition operator by multiplication operator
  - The original test data are then run through the *mutants*
  - If test data detect all differences in mutants, then the mutants are said to be *dead*, and the test set is *adequate*





Different types of Mutants

- Stillborn mutants: Syntactically incorrect, killed by compiler, e.g., x = a ++ b
- Trivial mutants: Killed by almost any test case
- Equivalent mutant: Always acts in the same behavior as the original program, e.g., x = a + b and x = a – (-b)

- None of the above are interesting from a mutation testing perspective
- Those mutants are interesting which behave differently than the original program, and we do not have test cases to identify them (to cover those specific changes)

Example of an Equivalent mutant

Original program

```
int index=0;
while (...)
{
      . . .;
      index++;
      if (index==10)
          break;
}
```

A mutant

```
int index=0;
while (...)
{
      . . .;
      index++;
      if (index>=10)
          break;
}
```

Basic Ideas (I)

In Mutation Testing:

1. We take a program and a test suite generated for that program (using other test techniques)
2. We create a number of *similar* programs (mutants), each differing from the original in one small way, i.e., each possessing a fault
   - E.g., replacing an addition operator by a multiplication operator
3. The original test data are then run on the *mutants*
4. If test cases detect differences in mutants, then the mutants are said to be *dead (killed),* and the test set is considered *adequate*

Basic Ideas (II)

- A mutant remains *live* either
  - because it is equivalent to the original program (functionally identical although syntactically different – called an *equivalent mutant*) or,
  - the test set is inadequate to kill the mutant
- In the latter case, the test data need to be augmented (by adding one or more new test cases) to kill the *live* mutant
- For the automated generation of mutants, we use *mutation operators*, that is predefined program modification rules (i.e., corresponding to a fault model)
- Example mutation operators next…

A Simple Example

Original Function		With Embedded Mutants
int Min (int A, int B)		int Min (int A, int B)
int minVal;		int minVal;
{		{
minVal = A;		minVal = A;
if (B < A)	Δ1	minVal = B;
{		if (B < A)
minVal = B;	Δ2	if (B > A)
}	Δ3	if (B < minVal)
return (minVal);		{
} // end Min		minVal = B;
	Δ4	Bomb();
	Δ5	minVal = A;
	Δ6	minVal = failOnZero (B);
		}
		return (minVal);
		} // end Min

Delta's represent syntactic modifications. In fact, each of them will be embedded in a different program version, a mutant.

Example of Mutation Operators I

- Constant replacement
- Scalar variable replacement
- Scalar variable for constant replacement
- Constant for scalar variable replacement
- Array reference for constant replacement
- Array reference for scalar variable replacement
- Constant for array reference replacement
- Scalar variable for array reference replacement
- Array reference for array reference replacement
- Source constant replacement
- Data statement alteration
- Comparable array name replacement
- Arithmetic operator replacement
- Relational operator replacement
- Logical connector replacement
- Absolute value insertion
- Unary operator insertion
- Statement deletion
- Return statement replacement

Example of Mutation Operators II

Specific to object-oriented programming languages:

- Replacing a type with a compatible subtype (inheritance)
- Changing the access modifier of an attribute, a method
- Changing the instance creation expression (inheritance)
- Changing the order of parameters in the definition of a method
- Changing the order of parameters in a call
- Removing an overloading method
- Reducing the number of parameters
- Removing an overriding method
- Removing a hiding Field
- Adding a hiding field

Specifying Mutations Operators

- Ideally, we would like the mutation operators to be representative of (and generate) all realistic types of faults that could occur in practice.
- Mutation operators change with programming languages, design and specification paradigms, though there is much overlap.
- In general, the number of mutation operators is large as they are supposed to capture all possible *syntactic* variations in a program.
- Recent paper suggests random sampling of mutants can be used.
- Some recent studies seem to suggest that mutants are good indicators of test effectiveness (Andrews et al, ICSE 2005).

Mutation Coverage

- Complete coverage equals to killing all non-equivalent mutants (or random sample)
- The amount of coverage is also called "mutation score"
- We can see each mutant as a test requirement
- The number of mutants depends on the definition of mutation operators and the syntax/structure of the software
- Numbers of mutants tend to be large, even for small programs (hence random sampling)

A Simple Example (again)

Original Function		With Embedded Mutants
`int Min (int A, int B)`		`int Min (int A, int B)`
`   int minVal;`		`   int minVal;`
`{`		`{`
`   minVal = A;`		`   minVal = A;`
`   if (B < A)`	$\Delta 1$	`   minVal = B;`
`   {`		`   if (B < A)`
`      minVal = B;`	$\Delta 2$	`   if (B > A)`
`   }`	$\Delta 3$	`   if (B < minVal)`
`return (minVal);`		`   {`
`} // end Min`		`      minVal = B;`
	$\Delta 4$	`      Bomb();`
	$\Delta 5$	`      minVal = A;`
	$\Delta 6$	`      minVal = failOnZero (B);`
		`   }`
		`return (minVal);`
		`} // end Min`

Delta's represent syntactic modifications. In fact, each of them will be embedded in a different program version, a mutant.

Discussion of the Example

- Mutant 3 is equivalent as, at this point, `minVal` and `A` have the same value
- Mutant 1: In order to find an appropriate test case to kill it, we must
  1. Reach the fault seeded during execution (Reachability)
     - Always true (i.e., we can always reach the seeded fault
  1. Cause the program state to be incorrect (Infection)
     - A <> B
  3. Cause the program output and/or behavior to be Incorrect (Propagation)
     - (B<A) = false

Original Function

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

With Embedded Mutants

```
     int Min (int A, int B)
        int minVal;
     {
        minVal = A;
Δ1      minVal = B;
        if (B < A)
Δ2      if (B > A)
Δ3      if (B < minVal)
        {
            minVal = B;
Δ4          Bomb();
Δ5          minVal = A;
Δ6          minVal = failOnZero (B);
        }
     return (minVal);
     } // end Min
```

Assumptions

- What about more complex errors, involving several statements?
- Let's discuss two assumptions:
  - Competent programmer assumption: They write programs that are nearly correct
  - Coupling effect assumption: Test cases that distinguish all programs differing from a correct one by only simple errors is so sensitive that they also implicitly distinguish more complex errors
- There is some empirical evidence of the above two hypotheses: Offutt, A.J., *Investigations of the Software Testing Coupling Effect*, ACM Transactions on Software Engineering and Methodology, vol. 1 (1), pp. 3-18, 1992.

Another Example

Specification:

- The program should prompt the user for a positive integer in the range 1 to 20 and then for a string of that length.
- The program then prompts for a character and returns the position in the string at which the character was first found or a message indicating that the character was not present in the string.

Code Chunk

```
…
found := FALSE;
i := 1;
while(not(found)) and (i <= x) do begin // x is the length
   if a[i] = c then
     found := TRUE
   else
     i := i + 1
end
if (found)
   print("Character %c appears at position %i");
else
   print("Character is not present in the string");
end
…
```

Mutation Testing Example: Test Set 1

Input				Expected Output (oracle)
x	**a[ ]**	**c**	**Response**	
25				**The input integer should be between 1 and 20**
1	**x**	**x**	**found**	**Character x appears at position 1**
1	**x**	**a**	**not found**	**Character is not present in the string**

Mutation Testing Example: Mutant 1 (for Test Set 1)

- Replace `Found := FALSE;` with `Found := TRUE;`
- Re-run original test data set
- Note: It is better in Mutation Testing to make only one small change at a time to avoid the danger of introduced faults with interfering effects (masking)
- Failure: "character *a* appears at position 1" instead of saying "character is not present in the string"
- Mutant 1 is killed (since Output <> Oracle)

```
…
found := FALSE; TRUE;
i := 1;
while(not(found)) and (i <= x) do begin
    if a[i] = c then
       found := TRUE
    else
       i := i + 1
end
if (found)
    print("Character %c appears at position %i");
else
    print("Character is not present in the string");
end
…
```

Mutation Testing Example: Mutant 2 (for Test Set 1)

- Replace `i:=1;` with `x:=1;`

```
Int i=1;
…
found := FALSE;
i := 1; x := 1;
while(not(found)) and (i <= x) do begin
    if a[i] = c then
       found := TRUE
    else
       i := i + 1
end
if (found)
    print("Character %c appears at position %i");
else
    print("Character is not present in the string");
end
…
```

- Will our original test data (test set 1) reveal the fault?
  - No, our original test data set fails to reveal the fault (because the x value was 1 in the second test case of test set 1)
- As a result of the fault, only position 1 in string will be searched for. So what should we do?
- In our test set, we need to increase our input string length and search for a character further along it
- We modify the test set 1 and create a new test set 2 (next) so as
  - To preserve the effect of earlier tests
  - To make sure the live mutant (#2) is killed

Mutation Testing Example: Test Set 2

Input				Expected Output
x	a	c	Actual output Response	
25				Input Integer between 1 and 20
1	x	x	found	Character x appears at position 1
1	x	a	not found	Character does not occur in string
3	xCv	v	Not found	Character v appears at position 3 (this test case will kill the mutant in the previous slide)

Mutation Testing Example: Mutant 3 (for Test Set 2)

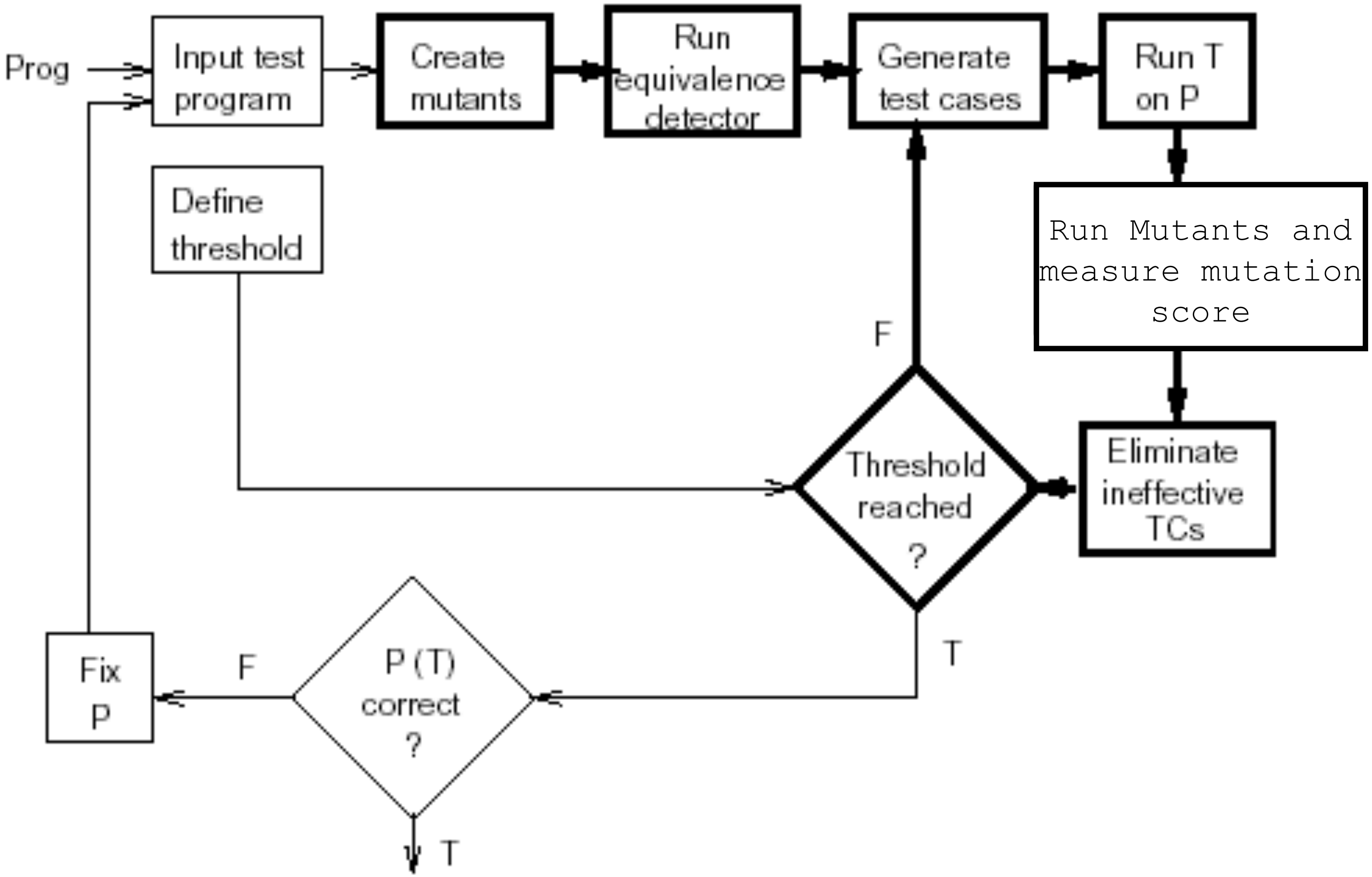
- `i := i + 1;` is replaced with `i:= i +2;`
- Again, our test data (test set 2) fails to kill the mutant
- We must augment the test set 2 and create a new test set 3 (next) to search for a character in the middle of the string
- With the new test set, mutant 3 can be killed
- Many other changes could be made on this short piece of code, e.g., changing array reference, changing the <= relational operator

```
…
found := FALSE;
i := 1;
while(not(found)) and (i <= x) do begin
    if a[i] = c then
       found := TRUE
    else
       i := i + 1 2
end
if (found)
    print("Character %c appears at position %i");
else
    print("Character is not present in the string");
end
…
```

Mutation Testing Example: Test Set 3

Input				Expected Output
x	a	c	Response	
25				Input Integer between 1 and 20
1	x	x	found	Character x appears at position 1
1	x	a	not found	Character does not occur in string
3	xCv	v	Found	Character v appears at position 3
3	xCv	C	Not found	Character C appears at position 2 (this test case will kill the mutant in the previous slide)

Mutation Testing Process



Jeff Offutt, A Practical System for Mutation Testing: Help for the Common Programmer, 1994

Mutation Testing: Discussion

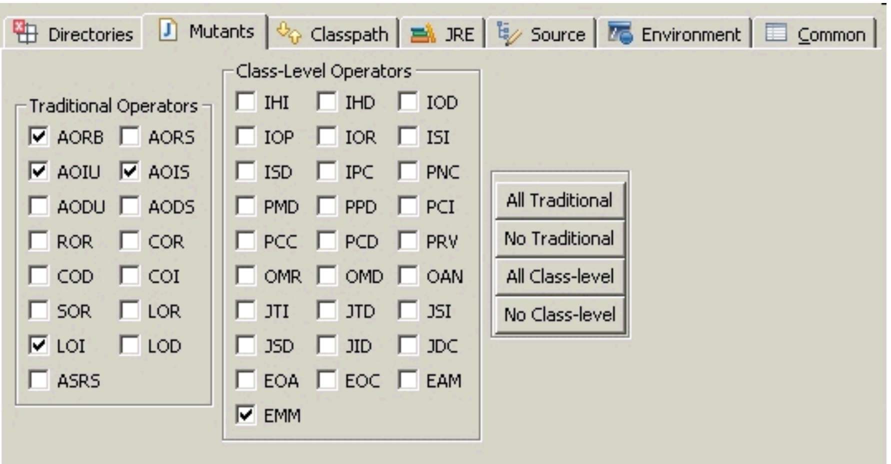
- It measures the quality of test cases
- A tool's slogan: "Jester - the JUnit test tester".
- It provides the tester with a clear target (mutants to kill)
- Mutation testing can also show that certain kinds of faults are unlikely (those specified by the fault model), since the corresponding test case will not fail
- It does force the programmer to inspect the code and think of the test data that will expose certain kinds of faults
- It is computationally intensive, a possibly very large number of mutants is generated: random sampling, selective mutation operators (Offutt)
- Equivalent mutants are a practical problem: It is in general an undecidable problem
- Probably most useful at unit testing level

Mutation Testing: Other Applications

- Mutation operators and systems are also very useful for assessing the effectiveness of test strategies – they have been used in a number of case studies
  - Define a set of realistic mutation operators
  - Generate mutants (automatically)
  - Generate test cases according to alternative strategies
  - Assess the *mutation score* (percentage of mutants killed)
- In our discussion, we saw mutation operators for source code (body)
- There are also works on
  - **Mutation operators for module interfaces (aimed at integration testing)**
  - **Mutation operators on specifications: Petri-nets, state machines, … (aimed at system testing)**

Mutation Testing Tools and Some Key Pointers

- Tools
  - **MuClipse: perhaps the best tool out there…**



  - **Jester: A Mutation Testing tool in Java (Open Source)**
  - **Pester: A Mutation Testing tool in Python (Open Source)**
  - **Nester: A Mutation Testing tool in C# (Open Source)**
  - http://www.parasoft.com/jsp/products/article.jsp?articleId=291

- Pointers:
  - http://en.wikipedia.org/wiki/Mutation_testing
  - http://www.mutationtest.net/
  - http://www.dcs.kcl.ac.uk/pg/jiayue/repository/