Mutation Testing for programs

Mutation testing for software artifacts: An overview

	For programs	Integration	Specifications	Input space
BNF grammar **Summary**	Programming languages Compilers	-	Algebraic specification s	Input languages (including XML) Input space testing
Mutation **Summary**	Programs Mutates program s	Programs Tests integration	FSMs Model checkin g	Input languages like XML Error checking

Program-based mutation: Overview

- Begin with the program (ground string).
- Apply one or more suitable mutation operators (mutant).
  - Mutants must be programs that compile.
  - Mutants are *not* tests.
- Write tests to kill the mutant.

A simple example of mutation

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

A simple example of mutation, contd.

One mutation of the Min method:

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

```
int Min(int A, int B)
{
    int minVal;
   minVal = A;
Δ1 minVal = B;
    if (B<A)
    {
        minVal = B;
    }
    return(minVal);
}
```

A simple example of mutation, contd.

Six different mutations of the `Min` method. Results in six different programs, each with one mutation.

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

```
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);
}
```

Mutants of Min: Explained

- Mutants 1, 3 and 5 replace one variable reference with another.
- Mutant 2 changes a relational operator.
- Mutant 4 is a special mutation operator that causes a run-time failure as soon as the statement is reached.
- Mutant 6 is another special mutation operator. `failOnZero()` method causes a failure if the parameter is zero and does nothing if the parameter is not zero (just returns the value of the parameter).

Mutants of programs

- Exhaustive mutation operators are available for several programming languages to be used for unit and integration testing.
- The goal of mutation operators is to mimic programmer mistakes.
- Mutation operators have to be selected carefully to strengthen the quality of mutation testing.
- Test cases are effective if they result in the mutated program exhibiting a behavior different from the original program.
- Several different mutants are defined in the testing literature to precisely capture the quality of mutation.

Mutants: Variants

- Stillborn mutant: Mutants of a program result in *invalid* programs that cannot even be compiled. Such mutants should not be generated.
- Trivial mutant: A mutant that can be killed by almost any test case.
- Equivalent mutant: Mutants that are functionally equivalent to a given program. No test case can kill them.
- Dead mutant: Mutants that are valid and can be killed by a test case. These are the only useful mutants for testing.

Killing a mutant: Refined notions

- Killing mutants, as defined earlier, might make it difficult to create tests all the time.
- Recap: Killing mutants: Given a mutant $m \in M$ for a ground string program P and a test t, t is said to kill m iff the output of t on P is different from the output of t on m.
- It *may not* be necessary to see the change only through an output all the time.
  - Just reachability and infection are enough.
  - Propagation may not be necessary.
- Programmers who do mutation testing for their own unit testing can have other means of observing the change in behavior.
- We refine killing a mutant and coverage to reflect this change.

Killing a mutant and coverage: Refined notions

- Strongly killing mutants: Given a mutant $m \in M$ for a ground string program P and a test t, t is said to strongly kill m iff the output of t on P is different from the output of t on m.
- Strong Mutation Coverage (SMC): For each $m \in M$, TR contains exactly one requirement to strongly kill m.
- Weakly killing mutants: 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 iff the state of the execution of P on t is different from the state of execution of m immediately after l.
- Weak Mutation Coverage (SMC): For each $m \in M$, TR contains exactly one requirement to weakly kill m.

Min method: Mutant 1

Consider the first mutant of Min method:

- Reachability: Always satisfied (True) as it is on the first statement.
- Infection: Value of A and B must be different; $A \; /= \; B$.
- Propagation: Mutation version of Min must return an incorrect value, i.e., statement in the `if` block must not be executed; $(B < A) = \text{false}$.
- Full test specification: $\text{True} \wedge (A \; /= \; B) \wedge ((B < A) = \text{false}) \equiv (A \; /= \; B) \wedge (B \geq A) \equiv (B > A)$.
- Test case A=5, B=7 will kill mutant 1. Original program will return 5, mutated version will return 7.

Min method: Mutant 3

The third mutant of `Min` method is an equivalent mutant.

- Intuitively, `minVal` and `A` have the same value at that statement in the program, so replacing one with the other has no effect.
- Reachability is true; infection condition is (B < A) /= (B < minVal).
- It is also true that (minVal = A) (assertion).
- Simplifying, we get (A /= minVal) $\wedge$ (minVal = A), a contradiction. This means that no value satisfying the conditions exist.

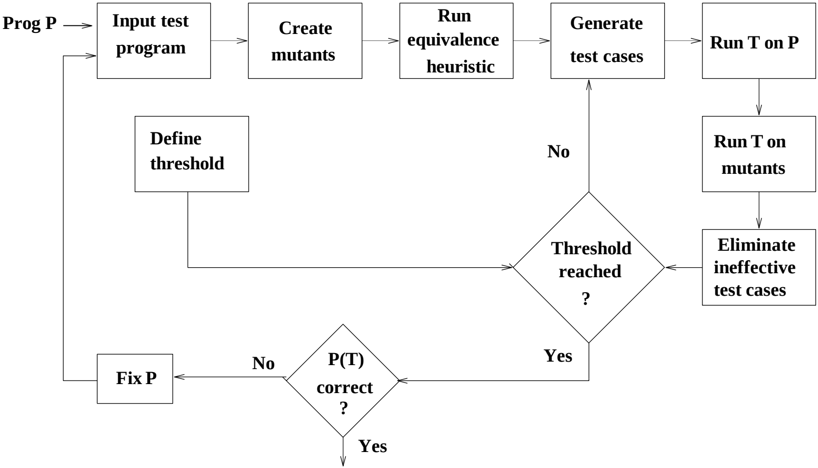
Another example

```
1   boolean isEven(int X)
2   {
3       if(X < 0)
4           X = 0-X;
Δ4          X = 0;
5       if(float)(X/2) == ((float)X)/2.0
6           return(true);
7       else
8           return(false);
9   }
```

Another example, contd.

- The mutant in line 4 is an example of weak killing.
  - Reachability: $(X < 0)$.
  - Infection: $(X \, / = 0)$.
- Consider the test case X=-6. Value of X after line 4:
  - In the original program: 6
  - In the mutated program: 0.
- Since 6 and 0 are both even, the decision at line 5 will return true for both the versions, so propagation is not satisfied.
- For strong killing, we need the test case to be an odd, negative integer.

Mutation testing: Process



Mutation testing for source code: Summary

- Mutation testing for source code is one of the most powerful forms of testing source code.
- We need to generate an *effective* set of test cases.
- Next lecture: Mutation operators for typical source code.

COURTESY:MEENAKSHI DSOUZA,IIIT ,BANGLORE