Fault Injection

Several key observations: a retrospective view

Control flow

Data flow

Error flow

Several key observations: a retrospective view

Control flow

<u>control flow graph (CFG)</u>

syntactically –based coverage does not reveal too much about the *semantics* of computations

can be executed with data that produce *coincidentally correct* output (despite the presence of faults in the code)

Data flow

Error flow

Several key observations: a retrospective view

Control flow

Data flow

def-use pairs

primarily *syntactic* in its nature, tries to emphasize exercising the semantic effect of a variable on other variables

Coincidental correctness is still a problem

Error flow

Several key observations: a retrospective view

Control flow
Data flow

•Some paths may be *infeasible*
(no data exists which cause their traversal).
Identifying infeasible paths can be difficult; in general it is an *undecidable problem*

•Difficulties to select test data

Error flow –creation and propagation of errors

Fault injection

Two main approaches:

•statistical testing

probability distribution generation of testing inputs; sampling; operational profiles

fault seeding

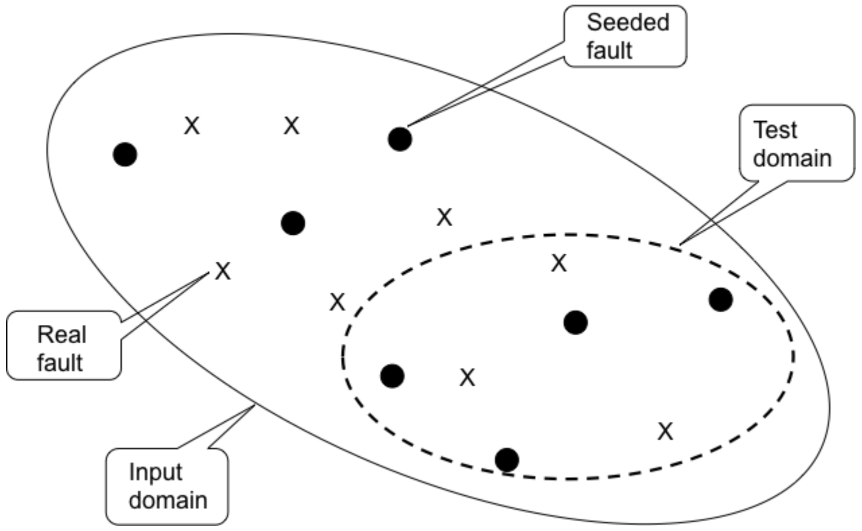
•Software mutation

Fault seeding (Mills)

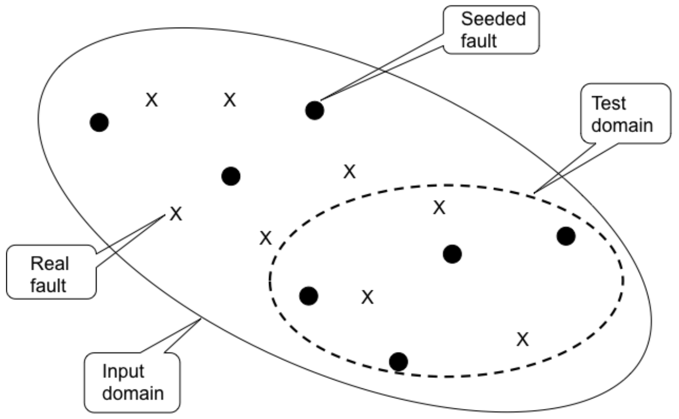
Originally developed to estimate the number of faults in a code

Motivates developers/testers:
• there is something to find
•They are not just looking for their own faults

Intentionally insert (seed) a known number of faults.
Testing team locates as many faults as possible



Fault seeding (Mills)



$$\boxed{\frac{\text{detected seeded faults (s)}}{\text{total seeded faults (S)}} = \frac{\text{detected nonseeded faults (n)}}{\text{total nonseeded faults (N)}}}$$

N =? Remaining faults N-n =?

Simple and intuitive approach

Validity of fault seeding. Default assumption?
<u>Seeded</u> faults are of the <u>same kind and complexity</u> as the <u>actual</u> faults

Fault seeding (Mills): expressing confidence (1)

Confidence level C (%) expresses likelihood that the software has a certain (N) number of faults

Code “seeded” with S faults.
Our claim is that the code has N actual faults.
Test the code until all S of the seeded faults have been found.
During this testing, “n” actual faults have been found as well

Confidence level C

$$\boxed{C = \frac{S}{S+N+1} \quad \text{if} \quad n \le N \qquad \text{and}\, C = 1\, \text{if} \quad n > N}$$

Fault seeding (Mills): expressing confidence (2)

Confidence level C

$$\boxed{C = \frac{S}{S+N+1} \quad \text{if} \quad n \le N \qquad \text{and } C = 1 \text{ if } \; n > N}$$

Fault seeding (Mills): expressing confidence (3)

Confidence level C

$$\boxed{C = \frac{S}{S+N+1} \quad \text{if} \quad n \le N \qquad \text{and}\, C = 1\, \text{if} \quad n > N}$$

Example

We claim N =0 (fault-free software)
Seed it with S =10; found all 10 faults without uncovering any indigenous fault
The confidence is 10./(10+1) = 10/11 =0.91

S=25 C= 0.96 S =50 C =0.98 S =100 C =0.99

Example

Required confidence C =0.98, N =0, how many seeded faults?
S = C/(1-C)
N =0 S=49

Fault seeding (Mills): expressing confidence (4)

Cannot predict the confidence level until all seeded faults are found

s – number of detected seeded faults

$$\boxed{C = \frac{\binom{S}{s-1}}{\binom{S+N+1}{N+s}} \quad \text{if } n \leq N \qquad\qquad C = 1 \quad \text{if } n > N}$$

When to stop testing?

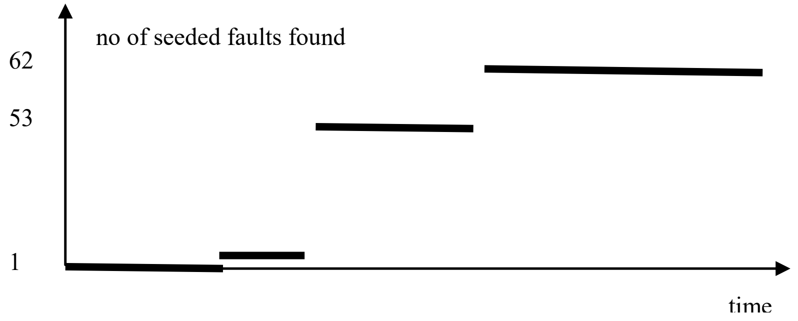
Fault seeding (Mills): expressing confidence (5)

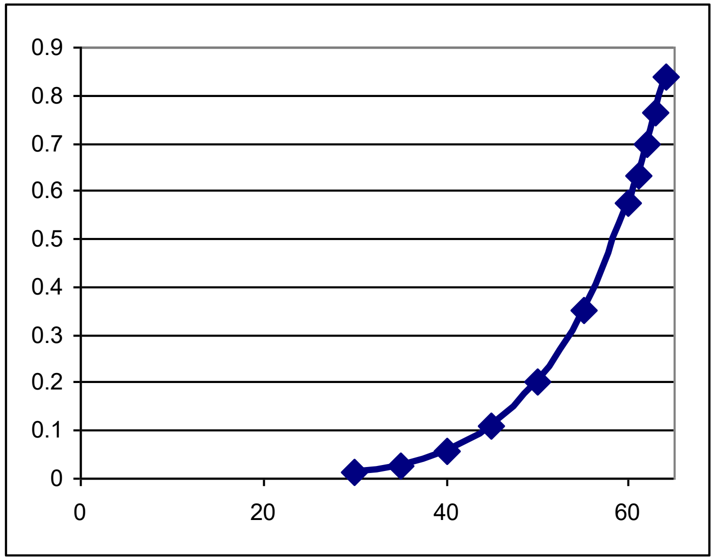
$$C = \frac{\binom{S}{s-1}}{\binom{S+N+1}{N+s}} \quad \text{if } n \leq N \qquad C = 1 \quad \text{if } n > N$$

Total number of seeded faults : 65
Number of nonseeded faults found: 3
Claim: 5 faults in software

CONFIDENCE LEVEL?

No of seeded faults found	conf
1	$6.9*10^{-9}$
53	0.283
62	0.697



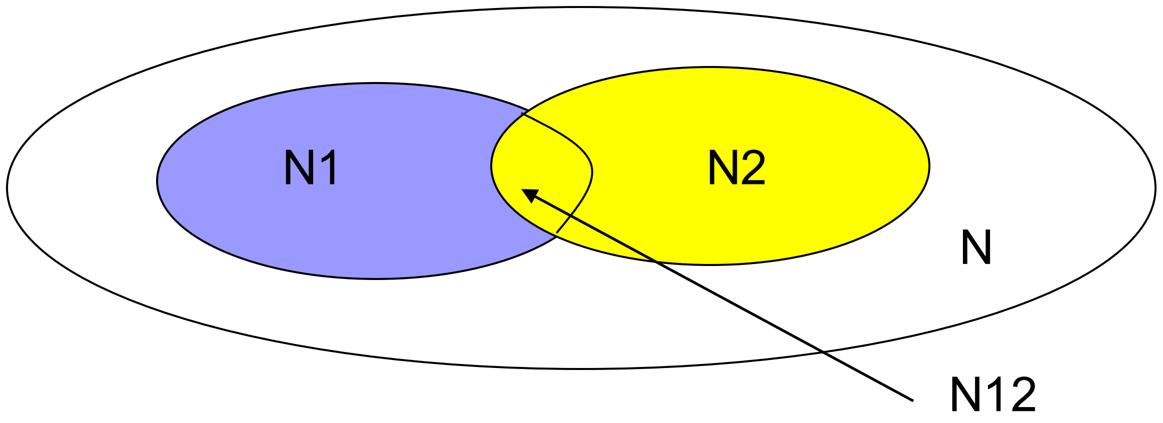


Independent tests (1)

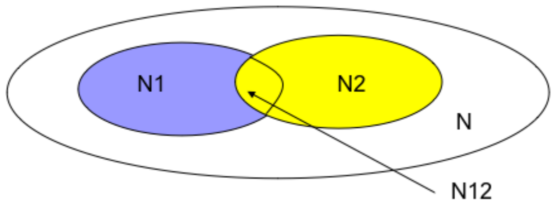
Testing done by two independent groups of testers using independent sets of test cases (capture-recapture model)

Discovered faults are collected and analyzed

N1- number of faults discovered by the first group
N2 – number of faults discovered by the second group
N12 – number of faults discovered by both groups



Independent tests (2) cross testing



Effectiveness of two groups of testers, E1 and E2

test team

tester E1 catches the same proportion of faults

in N2 as he/she catches in the remainder of the input space

$$\frac{N12}{N2} = \frac{N1}{N}$$

Determine N:

$$N = \frac{N1 * N2}{N12}$$

For instance

N1= 70 N2= 55 N12 = 30

N = (70*55)/30 = 128

Mutation testing

Mutation testing

What is a mutation?

- A mutation is a small change in a program.
- Such small changes are intended to model low level defects that arise in the process of coding systems.

Textbook Section 9.2.2

What is Mutation Testing?

- Mutation testing is a structural testing method aimed at assessing the **adequacy** of structural test suites.
- Is **NOT** a testing strategy like the strategies we studied so far

What is Mutation Testing?

Our goal is to gather data on the effectiveness of a particular test suite **T** in uncovering faults in a particular program P.

Mutation Testing: the Process

- We systematically apply mutations to the program P to obtain a sequence P_1, P_2,… P_n of mutants of P. Each mutant is derived by applying a single mutation operation to P.

- We run the test suite T on each of the mutants, T is said to **kill** mutant P_j if it detects a fault.

Mutation testing: mutants

Each mutant contains a **single fault-** differs from the original program by <u>one</u> mutation

A mutation is a single <u>syntactic</u> change that is made to a program

The goal is to cause the mutant program to fail, thus demonstrating the **effectiveness** of the test case

Test Case Adequacy

A test case is adequate if it is useful in detecting faults in a program

A test case can be shown to be **adequate** by finding (identifying) at least one mutant program that generates a different output than does the original program for that test case

If the original program and all mutant programs generate the same output, the test case is **inadequate**

Why does mutation testing work?

- Competent programmer hypothesis
- Coupling effect

The Competent Programmer hypothesis

•Programmers are generally very competent and do not create “random” programs

•For a given problem, a programmer, if mistaken will create a program that is very close to a correct one

•An incorrect program can be created from a correct one by making some minor change to the correct program

The Competent Programmer hypothesis

Semantic distance

Mapping of the program and its specification compared by their semantic distance defined as the number of inputs for which the two functions (program and specification) have different outputs

Competent programmer hypothesis:

code exhibits a *small semantic distance* from the specified function

Coupling Effect

Test data that distinguish all programs differing from a correct one by only simple errors is so sensitive that they also *implicitly* distinguish more complex errors

In other words:

- Simple faults compound in more complex faults
- Detecting simple atomic faults will lead to the detection of more complex faults

Fault→Failure

Fault-Failure /Success model [Morell & Hamlet]

A fault in a program P will produce a failure for a given input iff all of the following hold:

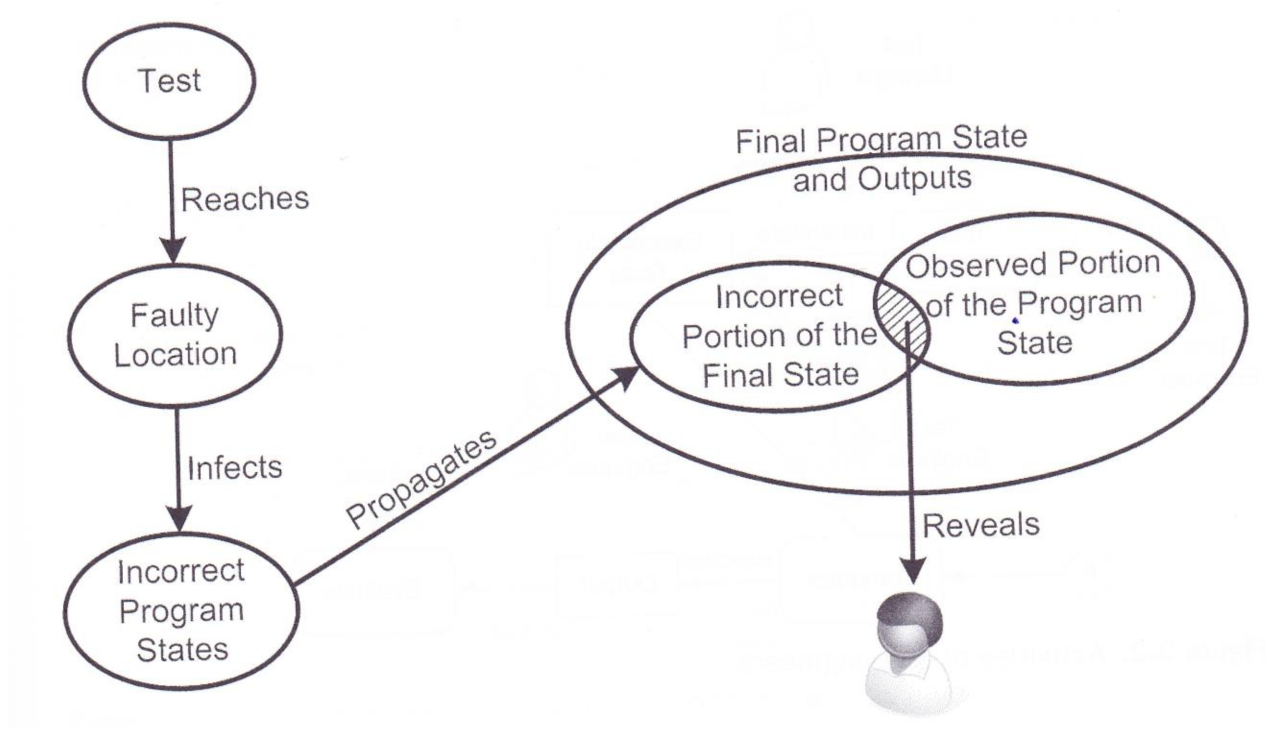
1. The fault is executed
2. Execution of the fault infects the subsequent data state
3. The infection propagates and causes failure

A faulty program P will produce *coincidental correctness* for a given input iff for each fault in P one of the following holds:

1. The fault is not executed
2. The execution of the fault does not infect the data state
3. The infection is cancelled by subsequent computation

RIPR model

Reachability, **i**nfection, **p**ropagation, **r**evealability


Test
Reaches
Faulty Location
Infects
Incorrect Program States
Propagates
Final Program State and Outputs
Incorrect Portion of the Final State
Observed Portion of the Program State
Reveals

textbook, Section 2.1

Types of mutation

- Value mutations
- Decision mutations
- Statement mutations

Value Mutations

these mutations involve changing the values of constants or parameters (by adding or subtracting values etc).

E.g. loop bounds – being one out on the start or finish is a very common error.

Value Mutations

- We attempt to change values to reflect errors in reasoning about programs.

- Typical examples are:
  - Changing values to one larger or smaller (or similar for real numbers).
  - Swapping values in some initialisation.

- The most common approach is to change constants by one in an attempt to generate a one-off error (particularly common in accessing arrays).

```
public int Segment(int t[], int l, int u){
      // Assumes t is in ascending order, and l<u,
      // counts the length of the segment
      // of t with each element l<t[i]<u
      int k = 0;

      for(int i=0; i<t.length && t[i]<u; i++){
            if(t[i]>l){
                  k++;
            }
      }
      return(k);
}
```

Mutating to k=1 causes miscounting

Here we might mutate the code to read i=1, a test that would kill this would have the length 1 and have l < t[0] < u, then the program would fail to count t[0] and return 0 rather than 1 as a result

Value Mutations- coverage

Coverage criterion:

Here we might want to perturb all constants in the program or unit at least once or twice.

Decision Mutations

involve modifying conditions to reflect potential slips and errors in the coding of <u>conditions</u> in programs.

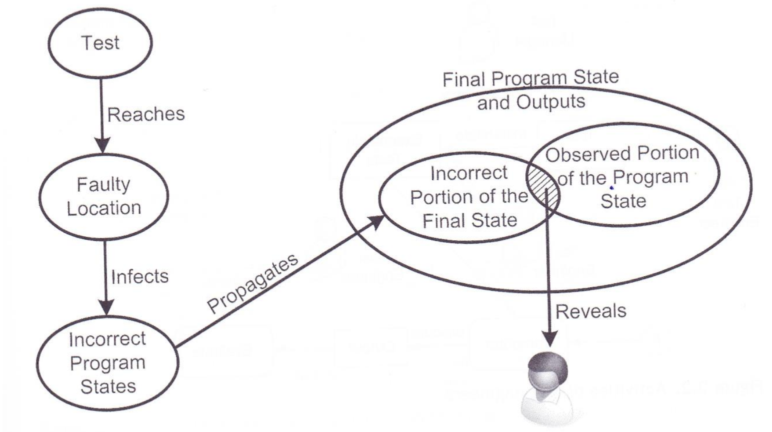
Decision Mutations

Typical examples are:

- Modelling “one-off” errors by changing < to <= or vice versa (this is common in checking loop bounds).
- Modelling confusion about larger and smaller, so changing > to < or vice versa.
- Getting parenthesisation wrong in logical expressions e.g. mistaking precedence between && and ||

RIPR model

Reachability, **i**nfection, **p**ropagation, **r**evealability


Test
Reaches
Faulty Location
Infects
Incorrect Program States
Propagates
Final Program State and Outputs
Incorrect Portion of the Final State
Observed Portion of the Program State
Reveals

```
if (a && b) {
      c = 1;
} else {
      c = 0;
}
```

```
if (a || b) {
      c = 1;
} else {
      c = 0;
}
```

Test **reaches** mutated statement
Test data **infect** the state
Incorrect state (value of c) **propagates**
value of c is **observed**

textbook, Section 2.1

```
public int Segment(int t[], int l, int u){
      // Assumes t is in ascending order, and l<u,
      // counts the length of the segment
      // of t with each element l<t[i]<u
      int k = 0;

      for(int i=0; i<t.length && t[i]<u; i++){
            if(t[i]>l){
                  k++;
            }
      }
      return(k);
}
```

Mutating to t[i]>u will cause miscounting if t[0]<u

We can model "one-off" errors in the loop bound by changing this condition to i<=t.length - provided array bounds are checked exactly this will provoke an error on every execution.

Decision Mutations

Coverage Criterion:

- We might consider one mutation for each condition in the program.
- Alternatively we might consider mutating all relational operators (and logical operators e.g. replacing || by && and vice versa)

Statement Mutations

these might involve deleting certain lines to reflect omissions in coding or swapping the order of lines of code. There are other operations e.g. changing operations in arithmetic expressions. A typical omission might be to omit the increment of some variable in a while loop.

Statement Mutations

Typical examples include:

- Deleting a line of code
- Duplicating a line of code
- Permuting the order of statements.

```
public int Segment(int t[], int l, int u){
      // Assumes t is in ascending order, and l<u,
      // counts the length of the segment
      // of t with each element l<t[i]<u
      int k = 0;

      for(int i=0; i<t.length && t[i]<u; i++){
            if(t[i]>l){
                  k++;
            }
      }
      return(k)
}
```

Here we might consider deleting this statement (then count would be zero for all inputs, we might also duplicate this line in which case all counts would be doubled.

Statement Mutations

Coverage Criterion:

- We might consider applying this procedure to each statement in the program (or all blocks of code up to and including a given small number of lines).

Mutation testing - process

Execute each test case against each alive mutant

- if the output of the mutant differs from the output of the original program, the mutant is considered incorrect and is killed

Survival of mutants

<u>Functionally equivalent (or equivalent)</u> to the original program: cannot be killed

```
int getMax(int x, int y) {
int max
if (x>y) max =x;
 else
max =y;
return (max);
}
```

```
int getMax(int x, int y) {
        int max
        if (x>y) max =x*1;
         else
        max =y;
        return (max);
        }
```

Survival of mutants

Killable: test cases are insufficient to kill the mutant. New test cases must be created

Mutation score

mutations score for a set of test cases is the percentage of non-equivalent mutants killed by the test data

$$\boxed{\text{mutation score} = \frac{D}{(N - E)}}$$

D – dead mutants

N – number of mutants

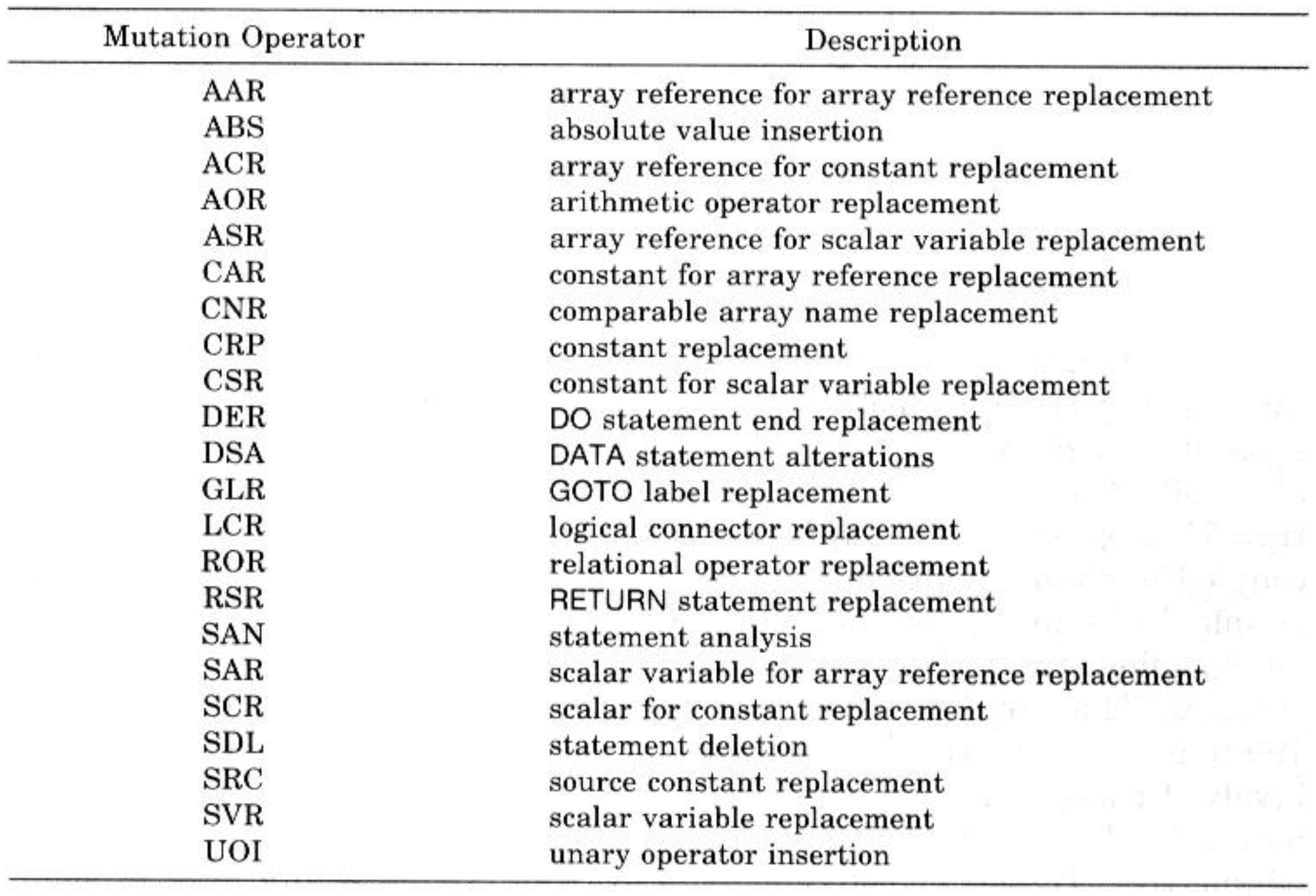
E – number of equivalent mutants

test cases are <u>mutation adequate</u> if its mutation score is 100%

Mothra

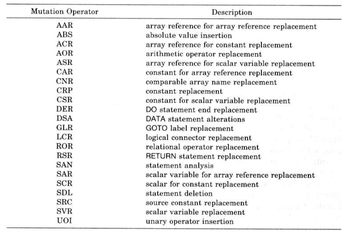
- Used in conjunction with Godzilla (metaphor of killing monsters)
- Tool to generate mutants
- Mutants are generated from an input set of error transforms
- Mothra originally used 22 transforms
- This generates many mutants, requires very large test sets, and takes a long time
- Is it possible to reduce the number of transforms while maintaining effectiveness?

Mothra’s mutation operators



Mutation Operator	Description
AAR	array reference for array reference replacement
ABS	absolute value insertion
ACR	array reference for constant replacement
AOR	arithmetic operator replacement
ASR	array reference for scalar variable replacement
CAR	constant for array reference replacement
CNR	comparable array name replacement
CRP	constant replacement
CSR	constant for scalar variable replacement
DER	DO statement end replacement
DSA	DATA statement alterations
GLR	GOTO label replacement
LCR	logical connector replacement
ROR	relational operator replacement
RSR	RETURN statement replacement
SAN	statement analysis
SAR	scalar variable for array reference replacement
SCR	scalar for constant replacement
SDL	statement deletion
SRC	source constant replacement
SVR	scalar variable replacement
UOI	unary operator insertion

Mothra’s mutation operators



Mutation Operator	Description
AAR	array reference for array reference replacement
ABS	absolute value insertion
ACR	array reference for constant replacement
AOR	arithmetic operator replacement
ASR	array reference for scalar variable replacement
CAR	constant for array reference replacement
CNR	comparable array name replacement
CRP	constant replacement
CSR	constant for scalar variable replacement
DER	DO statement end replacement
DSA	DATA statement alterations
GLR	GOTO label replacement
LCR	logical connector replacement
ROR	relational operator replacement
RSR	RETURN statement replacement
SAN	statement analysis
SAR	scalar variable for array reference replacement
SCR	scalar for constant replacement
SDL	statement deletion
SRC	source constant replacement
SVR	scalar variable replacement
UOI	unary operator insertion

ABS-absolute value insertion

```
x=3*a;

x=3*abs(a);     x=3*-abs(a);    x=abs(3*a);     x=-abs(3*a);
```

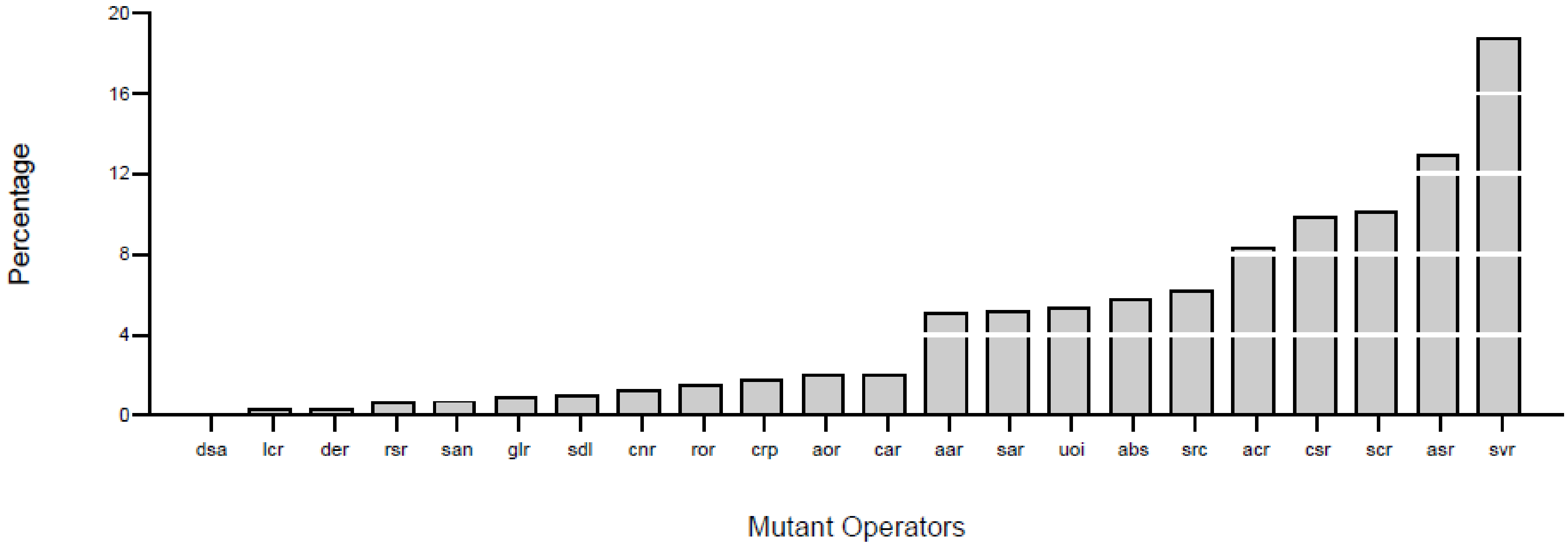
ROR- relational operator replacement

```
if(m>n);

if(m>=n); if(m<n); if(m<=n); if(m == n); if(m!=n);
if(false); if(true);
```

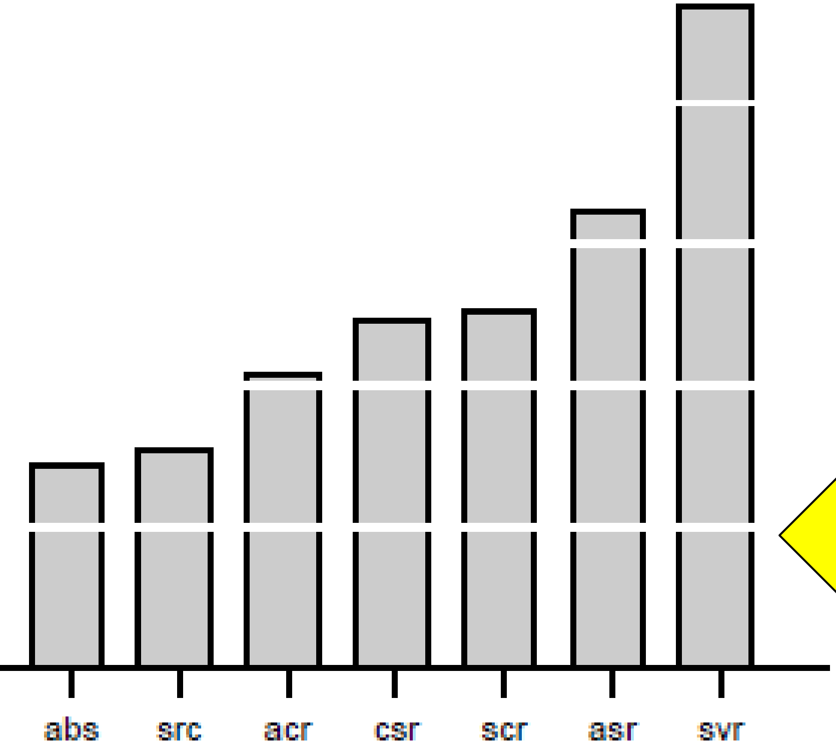
SVR- scalar variable replacement

```
x=a*b;

x=a*a; a=a*b; x=x*b; x=a*x; x=b*b; b=a*b;
```

% of mutants generated by mutation operators



% of mutants generated by mutation operators



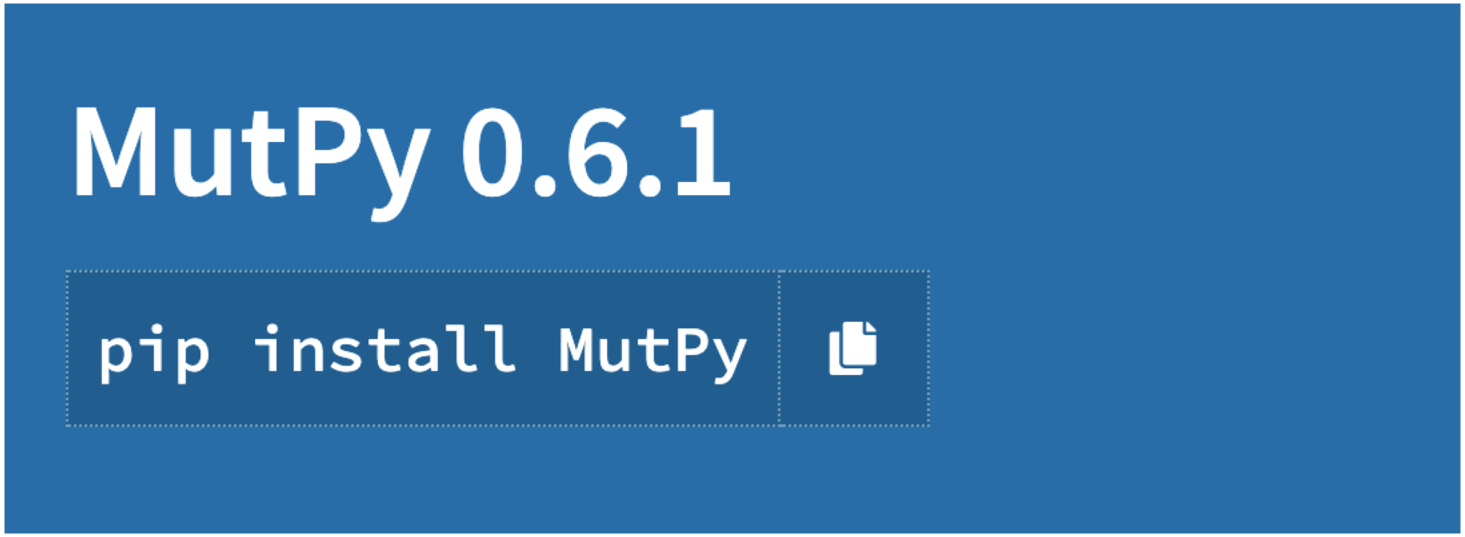
Scalar variable replacement

Mutation tools

MuJava mutation testing system for Java

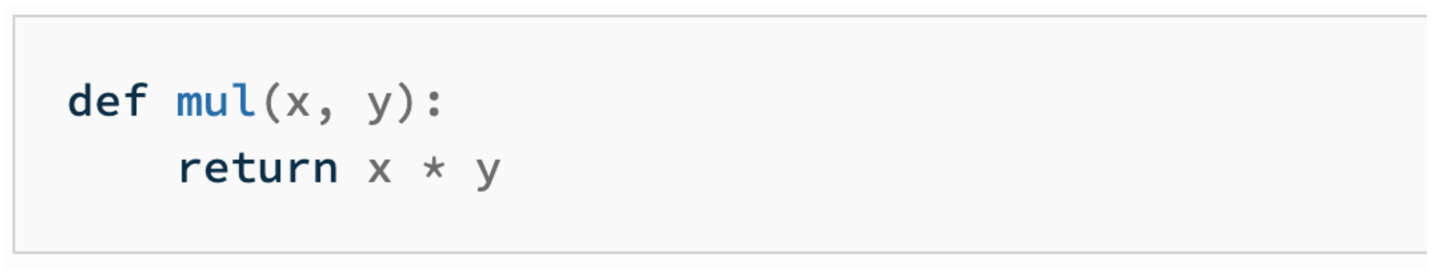
http://ise.gmu.edu/~ofut/mujava/

MuClipse Eclipse plugin for MuJava.
http://muclipse.sourceforge.net/

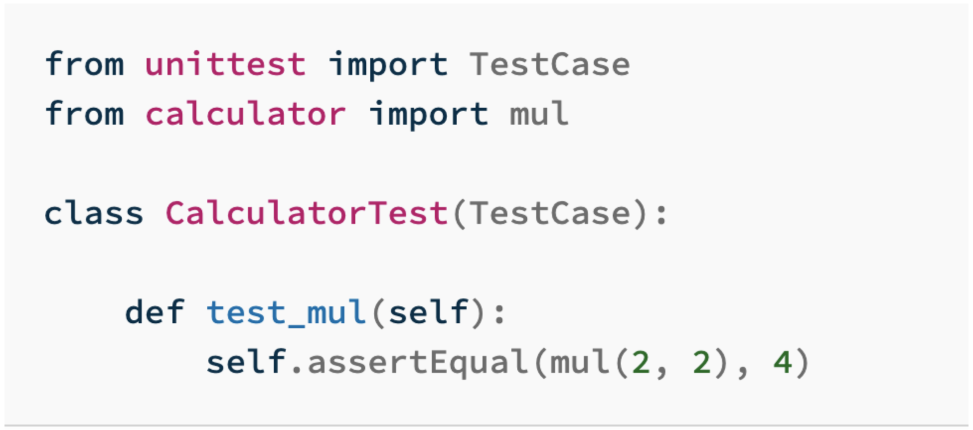


MutPy 0.6.1

```
pip install MutPy
```



```python
def mul(x, y):
    return x * y
```



```python
from unittest import TestCase
from calculator import mul

class CalculatorTest(TestCase):

    def test_mul(self):
        self.assertEqual(mul(2, 2), 4)
```

MutPy 0.6.1

```
$ mut.py --target calculator --unit-test test_calculator -m
```

```
[0.01437 s] killed by test_mul (test_calculator.CalculatorTest)
[*] Mutation score [0.21818 s]: 75.0%
   - all: 4
   - killed: 3 (75.0%)
   - survived: 1 (25.0%)
   - incompetent: 0 (0.0%)
   - timeout: 0 (0.0%)
```

```python
from unittest import TestCase
from calculator import mul

class CalculatorTest(TestCase):

    def test_mul(self):
        self.assertEqual(mul(2, 3), 6)
```

Mutation testing: quality

Test suite quality=

code coverage+ mutation score+ test redundancy

Observations (1)

- Mutations model low level errors in the mechanical software <u>development</u> process. Modelling <u>design</u> errors is much harder because they involve large numbers of coordinated changes throughout the program.

- Killing all mutants is very difficult (80-20%).
- Mutation adequate test suite will be adequate with respect to simple structural criteria of statement and branch coverage

Observations (2)

- Black-box test sets are poorer at killing mutants – we'd expect this because black-box tests are driven more by the requirements than by the need to cover statements, branches, etc.
- We could see mutation testing as a way of forcing more diversity on the development of test sets if we use a black-box approach as our primary test development approach.