Mutation Testing

W. Eric Wong
Department of Computer Science
The University of Texas at Dallas
ewong@utdallas.edu
http://www.utdallas.edu/~ewong

Speaker Biographical Sketch

11 27 2011

- Professor & Director of International Outreach
  Department of Computer Science
  University of Texas at Dallas
- Guest Researcher
  Computer Security Division
  National Institute of Standards and Technology (NIST)
- Vice President, IEEE Reliability Society
- Secretary, ACM SIGAPP (Special Interest Group on Applied Computing)
- Principal Investigator, NSF TUES (Transforming Undergraduate Education in Science, Technology, Engineering and Mathematics) Project
  - *Incorporating Software Testing into Multiple Computer Science and Software Engineering Undergraduate Courses*
- Founder & Steering Committee co-Chair for the SERE conference
  (*IEEE International Conference on Software Security and Reliability*)
  (http://paris.utdallas.edu/sere13)

Mutation Testing (1)

- The process of program development is considered as *iterative* whereby an initial version of the program is *refined* by making simple, or a combination of simple changes, towards the final version.

- Mutation testing is a *code-based test assessment* and improvement technique.
  - Can be extended to architecture (e.g., Statecharts) and design (e.g., SDL)

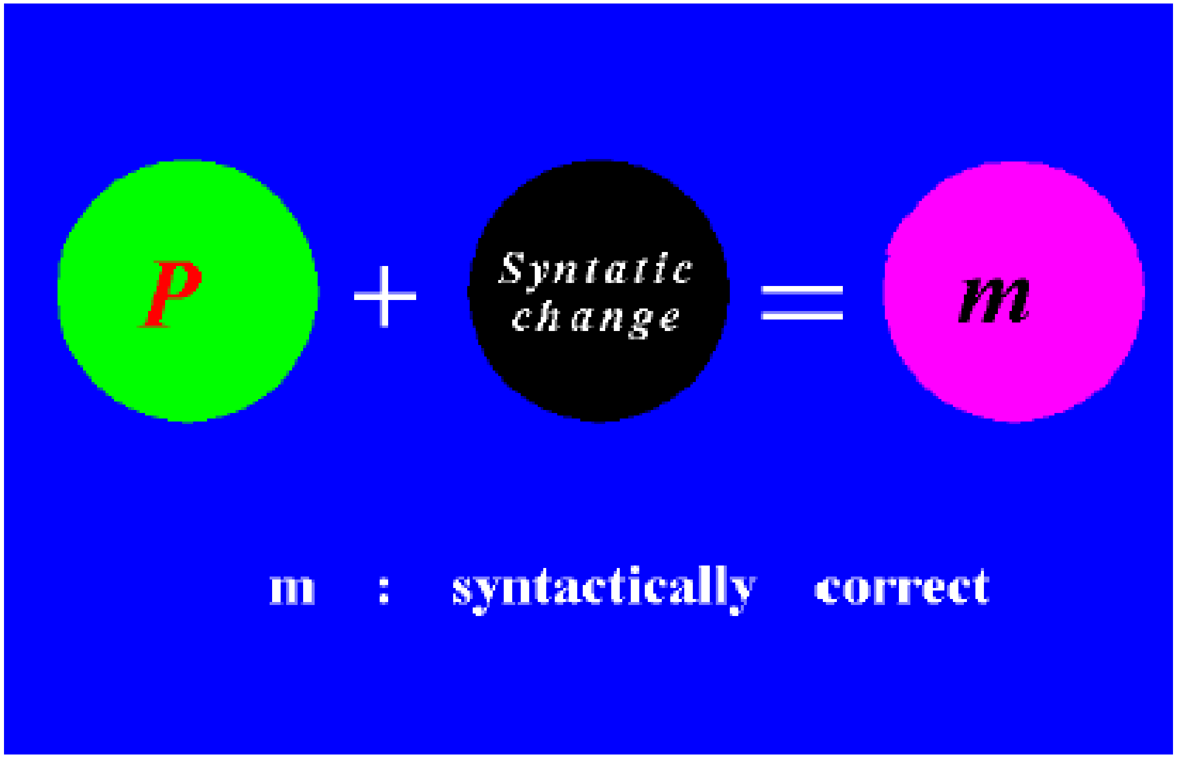
Mutation Testing (2)

- It relies on the *competent programmer hypothesis* which is the following assumption:
  - Given a specification a programmer develops a program that is either correct or differs from the correct program by a combination of simple errors

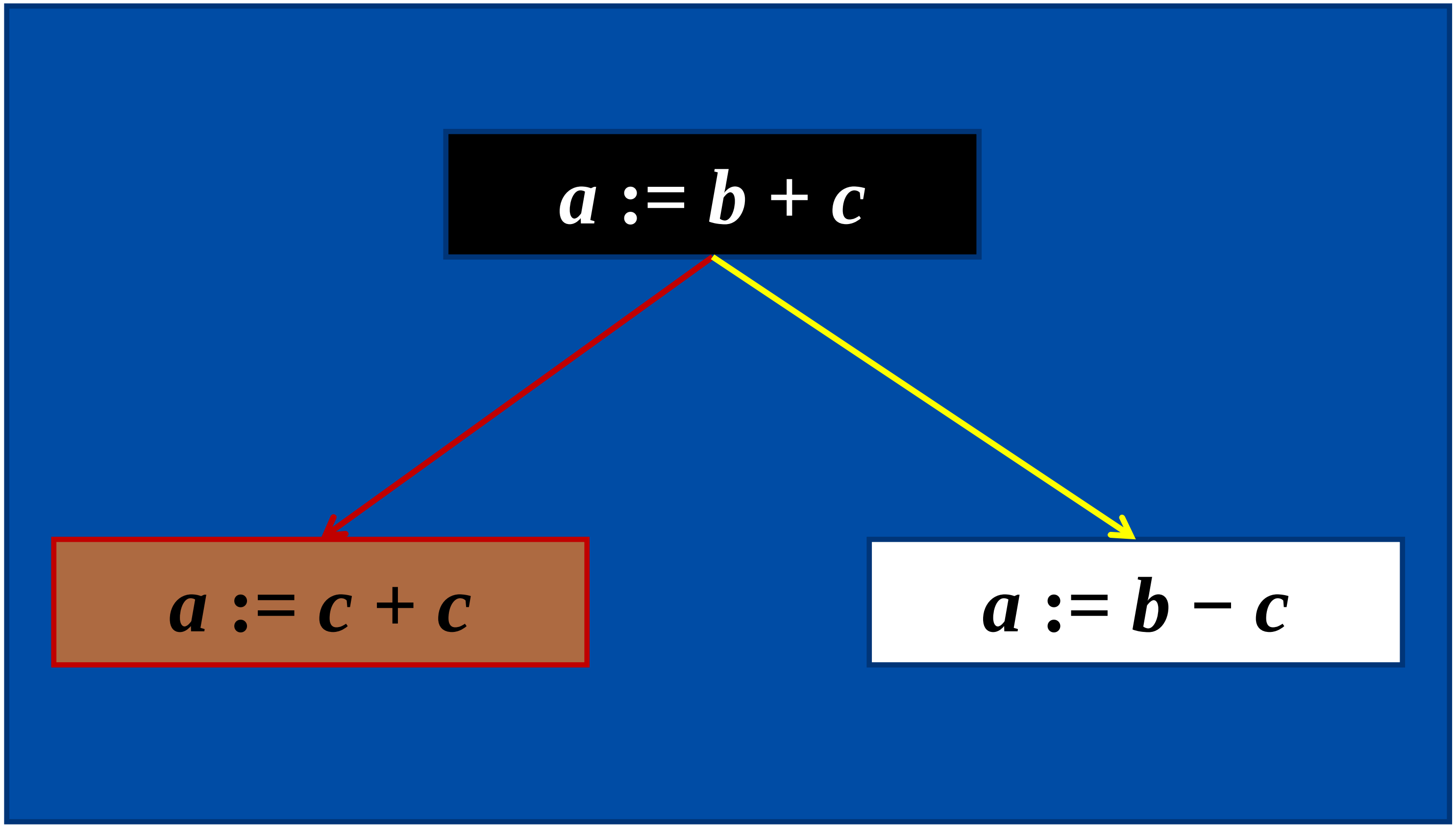
- It also relies on "*coupling effect*" which suggests that
  - Test cases that detect simple types of faults are sensitive enough to detect more complex types of faults.

Mutant (1)

- Given a program *P*, a *mutant* of *P* is obtained by making a *simple change* in *P*



Example (1)



$a := b + c$

$a := c + c$

$a := b - c$

Example (2)

Program

```
int x, y;
if (x != 0)
        y = 5;
else  z = z − x;
if (z > 1)
        z = z/x;
else
        z = y;
```

Mutant

```
int x, y;
if (x! = 0)
        y = 5;
else   z = z − x;
if (z > 1)
        z = z/&x;
else
        z = y;
```

Example (3)

Program

```
int x, y;
if (x! = 0)
        y = 5;
else z = z − x;
if (z > 1)
        z = z/x;
else
        z = y;
```

Mutant

```
int x, y;
if (x! = 0)
        y = 5;
else  z = z − x;
if (z < 1)
        z = z/x;
else
        z = y;
```

Order of Mutants

- First order mutants
  - One syntactic change
- Higher order mutants
  - Multiple syntactic changes
- Coupling effect

Type of Mutants (1)

- Distinguished mutants
- Live mutants
- Equivalent mutants
- Non-equivalent mutants

Type of Mutants (2)

- A mutant *m* is considered *distinguished* (or *killed*) by a test case $t \in T$ if

  $$P(t) \neq m(t)$$

  where $P(t)$ and $m(t)$ denote, respectively, the observed behavior of *P* and *m* when executed on test input *t*

- A mutant *m* is considered *equivalent* to *P* if

  $$P(t) = m(t)$$

  for any test case in the input domain

Distinguish a Mutant (1)

- Reachability
  - Execute the mutated statement

- Necessity
  - Make a state change

- Sufficiency
  - Propagate the change to output

Distinguish a Mutant (2)

```
Program $P$
read $a$
if ($a > 3$)
then
    $x = 5$
else
    $x = 2$
endif
print $x$
```

```
Program $m$
read $a$
if ($a \geq 3$)
then
    $x = 5$
else
    $x = 2$
endif
print $x$
```

Mutant m is distinguished by $a = 3$

Equivalent Mutant (1)

```
Program $P$
read $a$, $b$
$a = b$
$x = a + b$
print $x$
```

```
Program $m$
read $a$, $b$
$a = b$
$x = a + a$
print $x$
```

P is equivalent to *m*

Equivalent Mutant (2)

- Consider the following program *P*

```
int x, y, z;
scanf (&x, &y);
if (x>0)
     x = x + 1; z = x × (y − 1);
else
     x = x − 1; z = x × (y − 1);
```

- Here *z* is considered the output of *P*

Equivalent Mutant (3)

- Now suppose that a mutant of *P* is obtained by changing $x = x + 1$ to $x = abs(x) + 1$
- This mutant is equivalent to *P* as no test case can distinguish it from *P*

Mutation Score (1)

- During testing a mutant is considered *live* if it has not been distinguished or proven equivalent.

- Suppose that a total of $\mathcal{M}_t$ mutants are generated for program *P*

- The *mutation score* of a test set *T*, designed to test *P*, is computed as:

$$MS(P,T) = \frac{\mathcal{M}_k}{\mathcal{M}_t - \mathcal{M}_q}$$

  - $\mathcal{M}_k$ – number of mutants killed
  - $\mathcal{M}_q$ – number of equivalent mutants
  - $\mathcal{M}_t$ – total number of mutants

Mutation Score (2)

- Mutation score:

$$\frac{\text{Number of mutants distinguished}}{\text{Total number of non-equivalent mutants}}$$

- Data flow score:

$$\frac{\text{Number of blocks (decisions, p-uses, c-uses, all-uses) covered}}{\text{Total number of feasible blocks (decisions, p-uses, c-uses, all-uses)}}$$

Test Adequacy Criterion

- A test *T* is considered *adequate* with respect to the mutation criterion if its mutation score is 1
  - Equivalent mutants?
  - Which mutant operators are used?

- The number of mutants generated depends on *P* and the *mutant operators* applied on *P*

- A *mutant operator* is a rule that when applied to the program under test generates zero or more mutants

Mutant Operator (1)

- Consider the following program:

```
int abs (x);
int x;
{
    if (x ≥ 0) x = 0 − x;
    return x;
}
```

Mutant Operator (2)

- Consider the following rule:
  - Replace each relational operator in *P* by all possible relational operators excluding the one that is being replaced.

- Assuming the set of relational operators to be: {<, >, ≤, ≥, = =, !=}, the above mutant operator will generate *a total of 5 mutants of P*

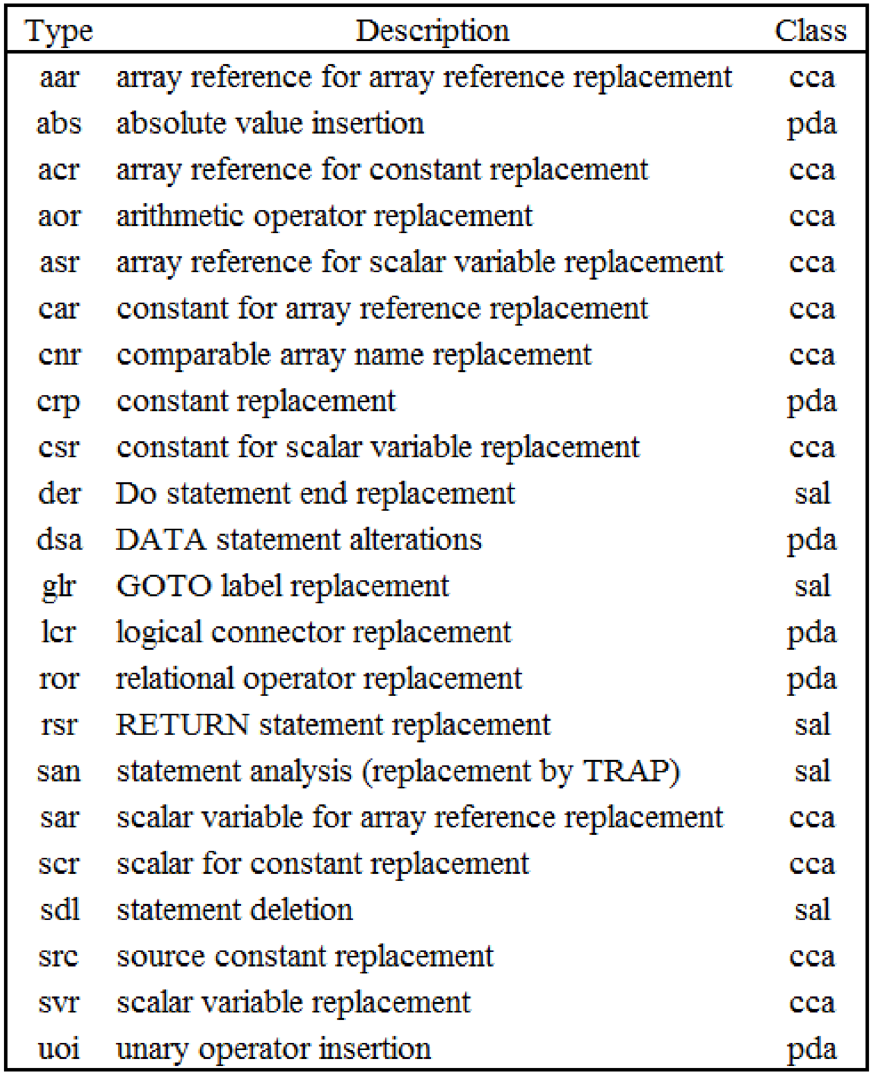
Mutant Operator **(3)**

- Mutation operators are *language dependent*

- For Fortran a total of 22 operators were proposed

- For C a total of 77 operators were proposed

Mutant Operators for Fortran **(1)**



Type	Description	Class
aar	array reference for array reference replacement	cca
abs	absolute value insertion	pda
acr	array reference for constant replacement	cca
aor	arithmetic operator replacement	cca
asr	array reference for scalar variable replacement	cca
car	constant for array reference replacement	cca
cnr	comparable array name replacement	cca
crp	constant replacement	pda
csr	constant for scalar variable replacement	cca
der	Do statement end replacement	sal
dsa	DATA statement alterations	pda
glr	GOTO label replacement	sal
lcr	logical connector replacement	pda
ror	relational operator replacement	pda
rsr	RETURN statement replacement	sal
san	statement analysis (replacement by TRAP)	sal
sar	scalar variable for array reference replacement	cca
scr	scalar for constant replacement	cca
sdl	statement deletion	sal
src	source constant replacement	cca
svr	scalar variable replacement	cca
uoi	unary operator insertion	pda

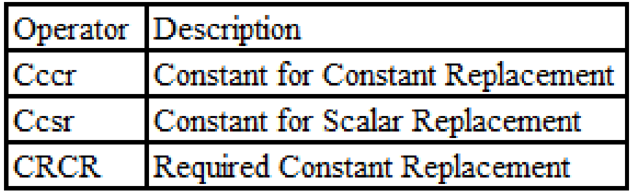
Mutant Operators for Fortran (2)

- *san*: replace each statement by TRAP
  (an instruction that causes the program to halt, killing the mutant)
  - Which code coverage-based criterion will also be satisfied by killing all the san mutants?

- *rsr*: replace each statement in a subprogram by RETURN

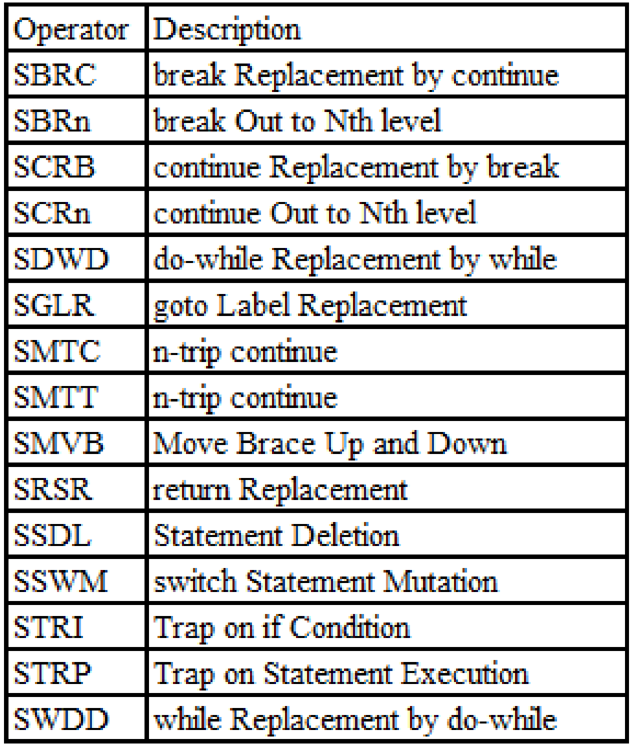
Mutant Operators for C **(1)**

Table 1: Constant Class Operators



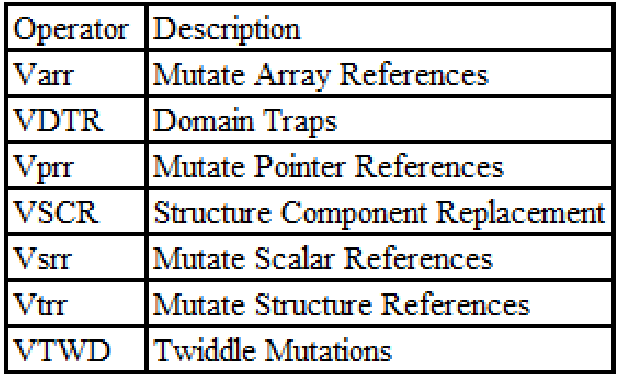
Operator	Description
Cccr	Constant for Constant Replacement
Ccsr	Constant for Scalar Replacement
CRCR	Required Constant Replacement

Table 2: Statement Class Operators



Operator	Description
SBRC	break Replacement by continue
SBRn	break Out to Nth level
SCRB	continue Replacement by break
SCRn	continue Out to Nth level
SDWD	do-while Replacement by while
SGLR	goto Label Replacement
SMTC	n-trip continue
SMTT	n-trip continue
SMVB	Move Brace Up and Down
SRSR	return Replacement
SSDL	Statement Deletion
SSWM	switch Statement Mutation
STRI	Trap on if Condition
STRP	Trap on Statement Execution
SWDD	while Replacement by do-while

Table 3: Variable Class Operators



Operator	Description
Varr	Mutate Array References
VDTR	Domain Traps
Vprr	Mutate Pointer References
VSCR	Structure Component Replacement
Vsrr	Mutate Scalar References
Vtrr	Mutate Structure References
VTWD	Twiddle Mutations

Mutant Operators for C (2)

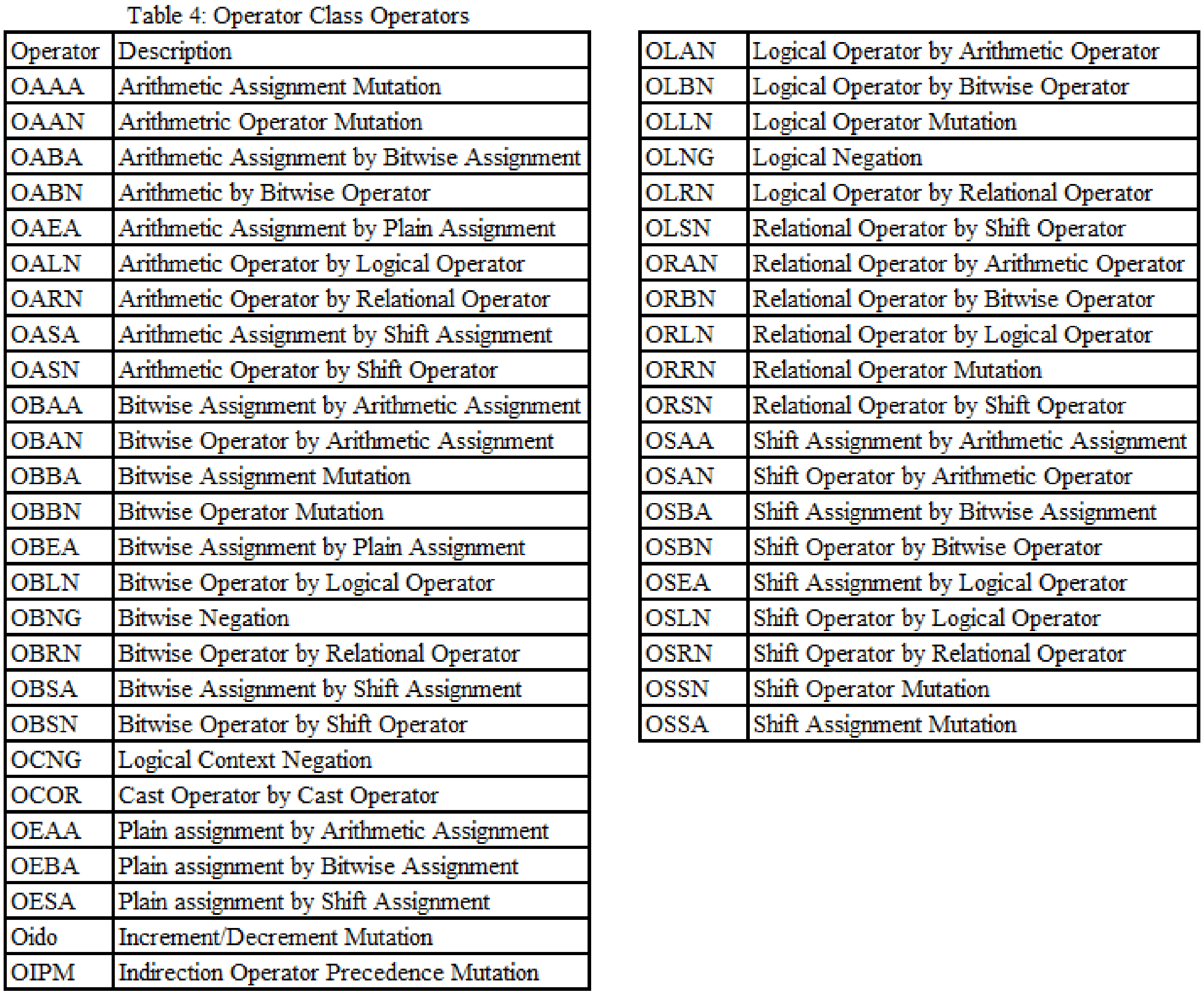


Table 4: Operator Class Operators

Operator	Description
OAAA	Arithmetic Assignment Mutation
OAAN	Arithmetic Operator Mutation
OABA	Arithmetic Assignment by Bitwise Assignment
OABN	Arithmetic by Bitwise Operator
OAEA	Arithmetic Assignment by Plain Assignment
OALN	Arithmetic Operator by Logical Operator
OARN	Arithmetic Operator by Relational Operator
OASA	Arithmetic Assignment by Shift Assignment
OASN	Arithmetic Operator by Shift Operator
OBAA	Bitwise Assignment by Arithmetic Assignment
OBAN	Bitwise Operator by Arithmetic Assignment
OBBA	Bitwise Assignment Mutation
OBBN	Bitwise Operator Mutation
OBEA	Bitwise Assignment by Plain Assignment
OBLN	Bitwise Operator by Logical Operator
OBNG	Bitwise Negation
OBRN	Bitwise Operator by Relational Operator
OBSA	Bitwise Assignment by Shift Assignment
OBSN	Bitwise Operator by Shift Operator
OCNG	Logical Context Negation
OCOR	Cast Operator by Cast Operator
OEAA	Plain assignment by Arithmetic Assignment
OEBA	Plain assignment by Bitwise Assignment
OESA	Plain assignment by Shift Assignment
Oido	Increment/Decrement Mutation
OIPM	Indirection Operator Precedence Mutation
OLAN	Logical Operator by Arithmetic Operator
OLBN	Logical Operator by Bitwise Operator
OLLN	Logical Operator Mutation
OLNG	Logical Negation
OLRN	Logical Operator by Relational Operator
OLSN	Relational Operator by Shift Operator
ORAN	Relational Operator by Arithmetic Operator
ORBN	Relational Operator by Bitwise Operator
ORLN	Relational Operator by Logical Operator
ORRN	Relational Operator Mutation
ORSN	Relational Operator by Shift Operator
OSAA	Shift Assignment by Arithmetic Assignment
OSAN	Shift Operator by Arithmetic Operator
OSBA	Shift Assignment by Bitwise Assignment
OSBN	Shift Operator by Bitwise Operator
OSEA	Shift Assignment by Logical Operator
OSLN	Shift Operator by Logical Operator
OSRN	Shift Operator by Relational Operator
OSSN	Shift Operator Mutation
OSSA	Shift Assignment Mutation

Mutation Testing Procedure **(1)**

- Given *P* and a test set *T*
  - Generate mutants
  - Compile *P* and the mutants
  - Execute *P* and the mutants on each test case
  - Determine equivalent mutants
  - Determine mutation score
  - If mutation score is not 1 then improve the test case and repeat from Step 3

Mutation Testing Procedure (2)

- In practice the above procedure is implemented incrementally

- One applies a few selected mutant operators to *P* and computes the mutation score with respect to the mutants generated

- Once these mutants have been distinguished or proven equivalent, another set of mutant operators is applied

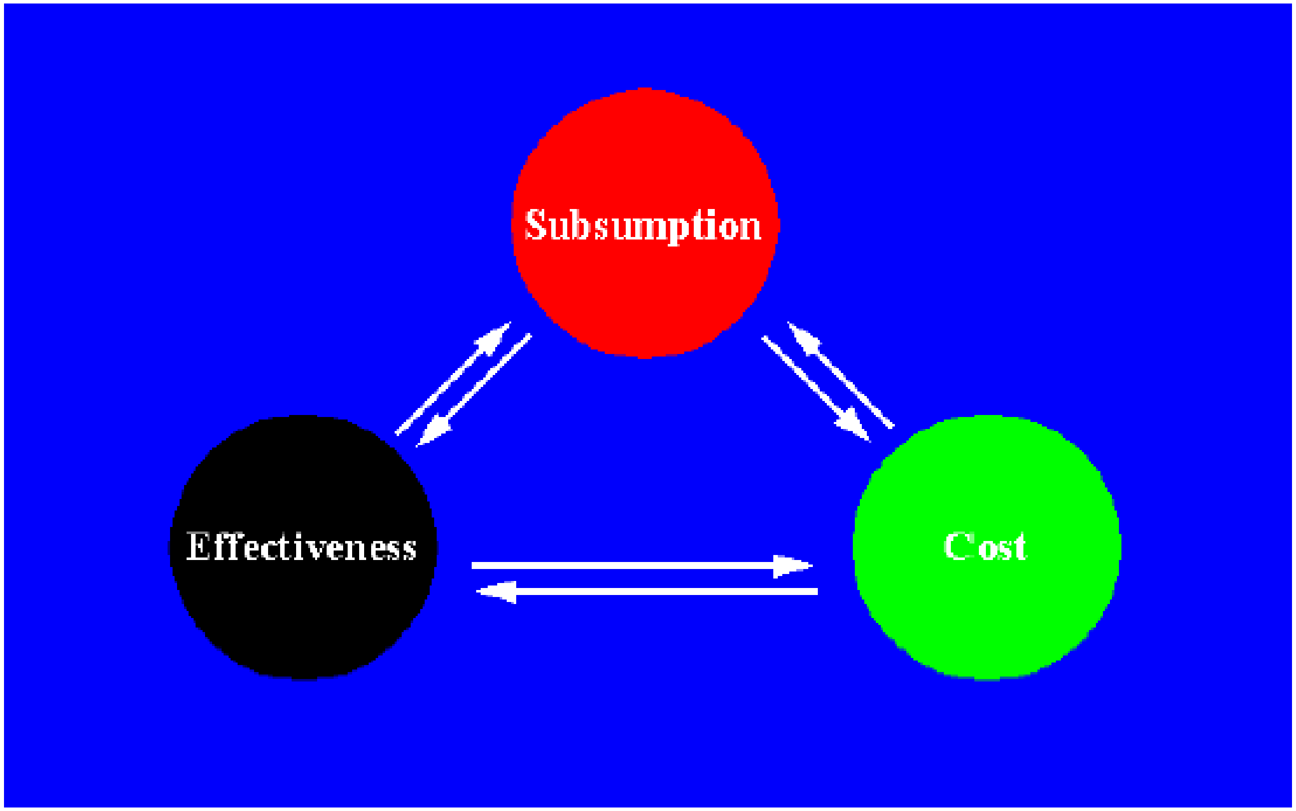
Mutation Testing Procedure (3)

- This procedure is repeated until either *all the mutants* have been exhausted or some external condition forces testing to stop

Tools for Mutation Testing

- *Mothra*: for Fortran, developed at Purdue, 1990
- *Proteum*: for C, developed at the University of São Paulo at São Paulo in Brazil.

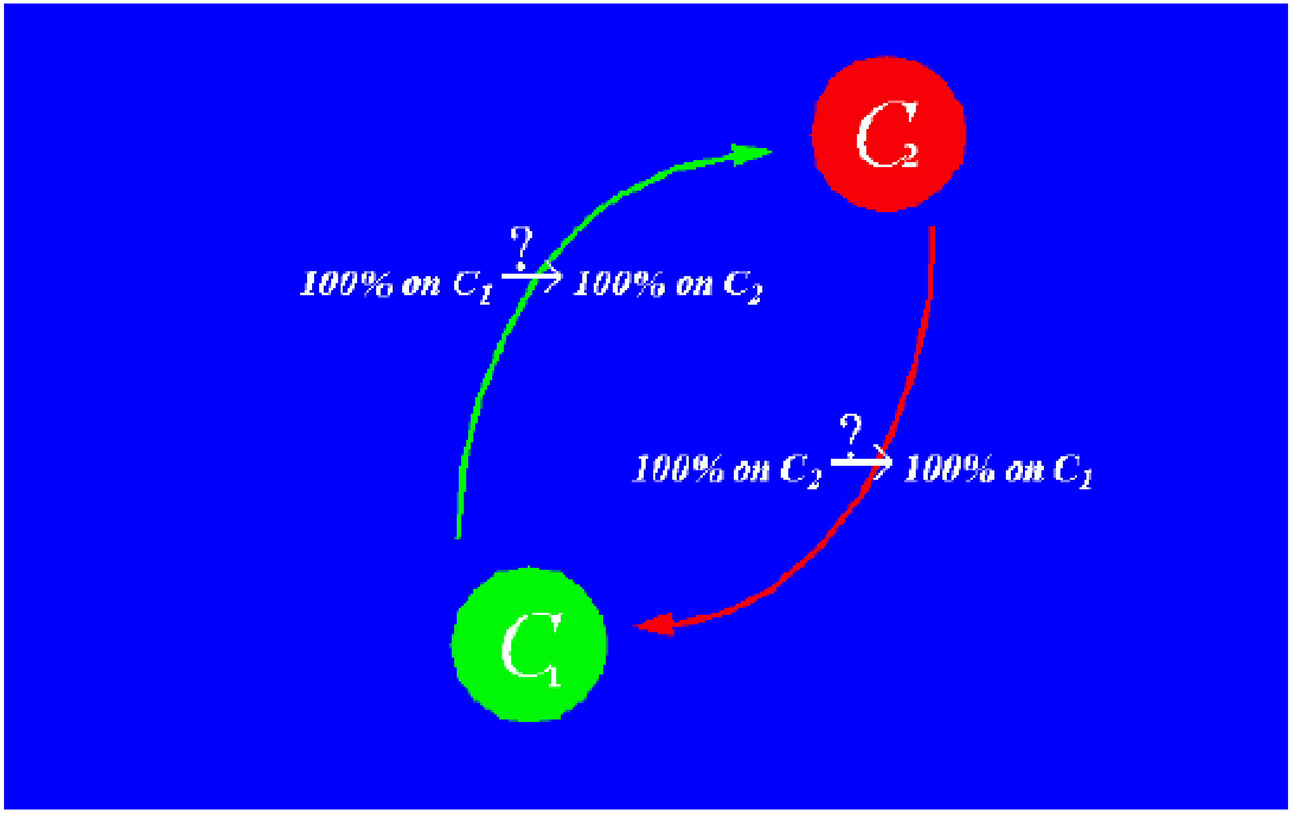
Comparison Criterion



Mutation Hypothesis

- More difficulty to satisfy
- More expensive
- More effective in fault detection

Subsumption


C_2
$100\% \text{ on } C_1 \xrightarrow{?} 100\% \text{ on } C_2$
$100\% \text{ on } C_2 \xrightarrow{?} 100\% \text{ on } C_1$
C_1

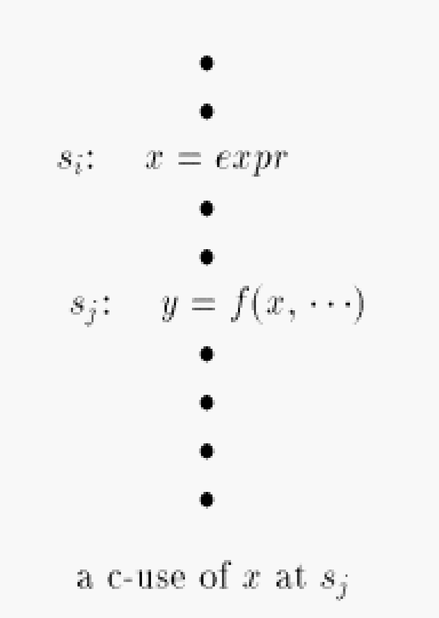
Program Classification

- SDSU (single definition, single use)
- SDMU (single definition, multiple uses)
- MDSU (multiple definitions, single use)
- MDMU (multiple definitions, multiple uses)

Program Classification

- SDSU
  - M (Mothra) subsumes AU, CU, and PU
- SDMU
  - M (Mothra) subsumes AU, CU, and PU
- MDSU (multiple definitions, single use)
  - M (Mothra) subsumes CU, but not PU, and AU
- MDMU
  - M (Mothra) does not subsume CU, PU, and AU

SDSU (1)



•

•

s_i: $x = expr$

•

•

s_j: $y = f(x, \cdots)$

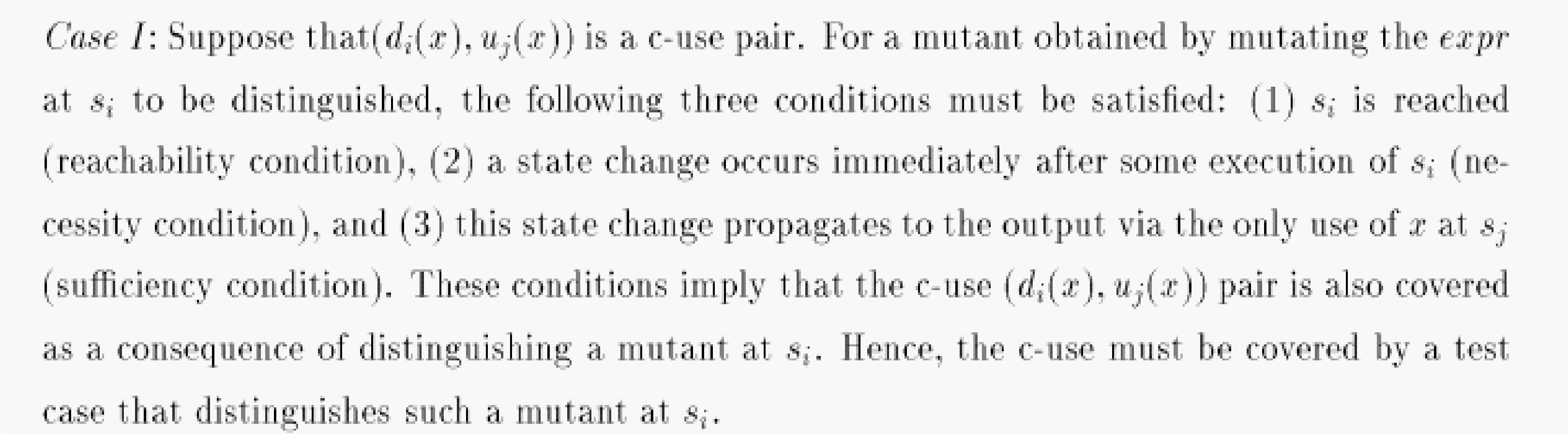
•

•

•

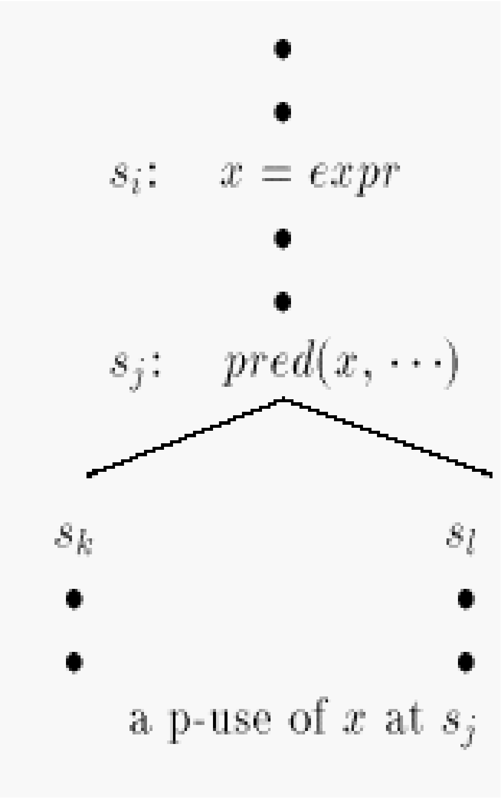
•

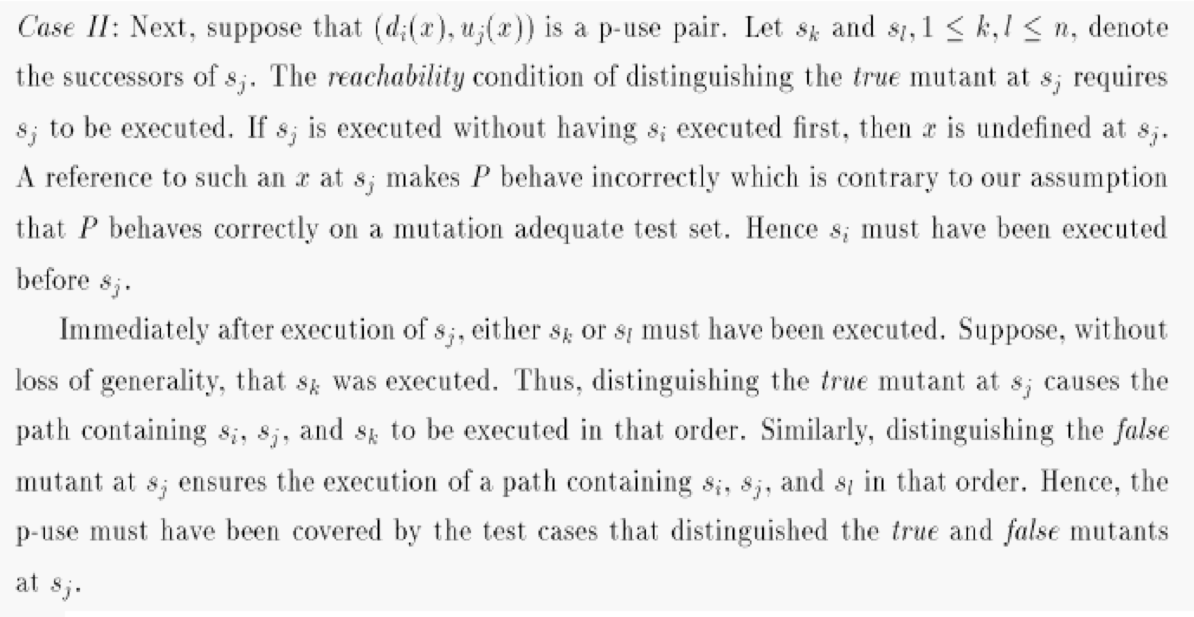
a c-use of x at s_j



Case I: Suppose that$(d_i(x), u_j(x))$ is a c-use pair. For a mutant obtained by mutating the $expr$ at s_i to be distinguished, the following three conditions must be satisfied: (1) s_i is reached (reachability condition), (2) a state change occurs immediately after some execution of s_i (necessity condition), and (3) this state change propagates to the output via the only use of x at s_j (sufficiency condition). These conditions imply that the c-use $(d_i(x), u_j(x))$ pair is also covered as a consequence of distinguishing a mutant at s_i. Hence, the c-use must be covered by a test case that distinguishes such a mutant at s_i.

SDSU (2)

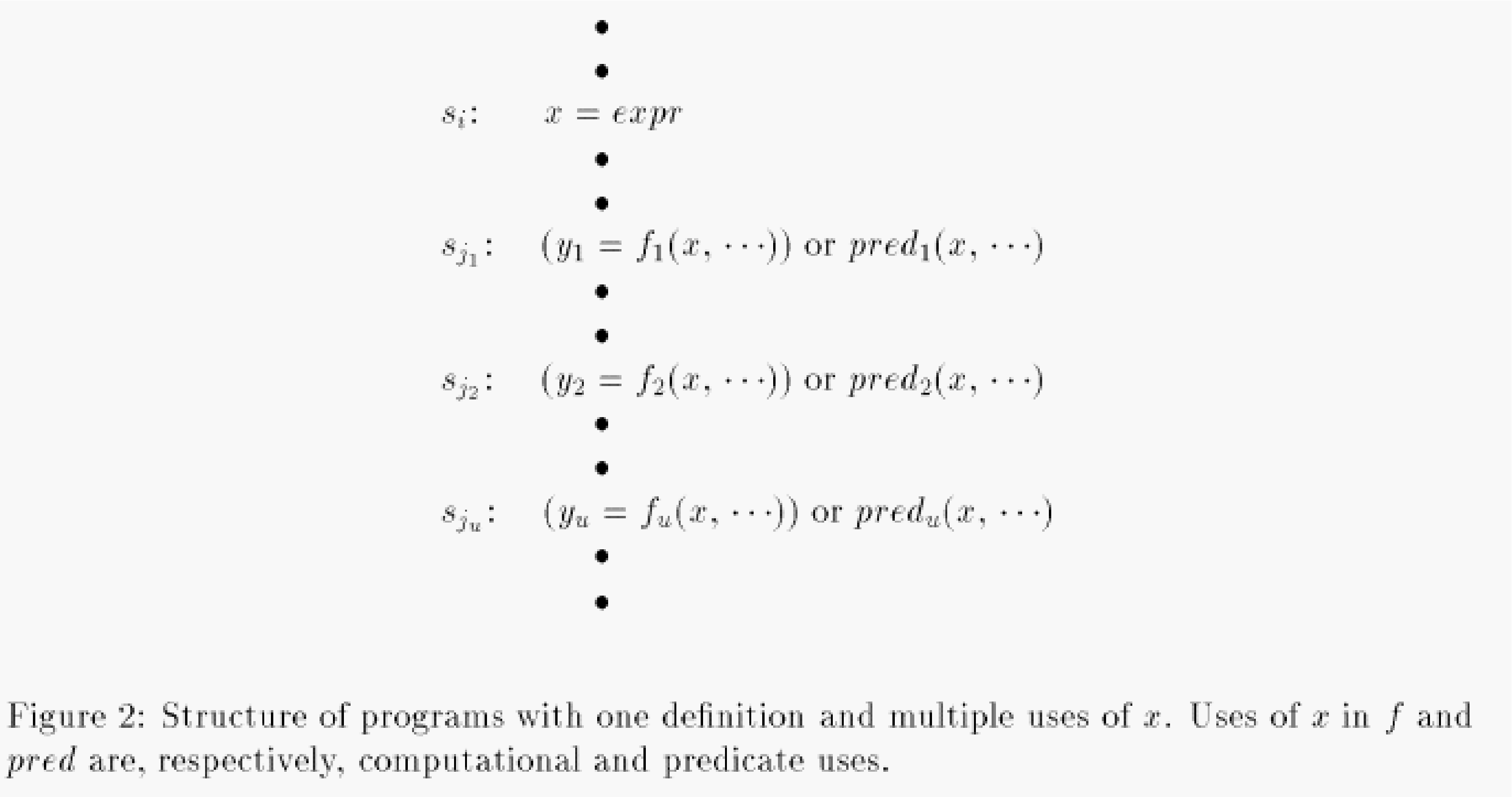




Case II: Next, suppose that $(d_i(x), u_j(x))$ is a p-use pair. Let s_k and $s_l, 1 \leq k, l \leq n$, denote the successors of s_j. The *reachability* condition of distinguishing the *true* mutant at s_j requires s_j to be executed. If s_j is executed without having s_i executed first, then x is undefined at s_j. A reference to such an x at s_j makes P behave incorrectly which is contrary to our assumption that P behaves correctly on a mutation adequate test set. Hence s_i must have been executed before s_j.

Immediately after execution of s_j, either s_k or s_l must have been executed. Suppose, without loss of generality, that s_k was executed. Thus, distinguishing the *true* mutant at s_j causes the path containing s_i, s_j, and s_k to be executed in that order. Similarly, distinguishing the *false* mutant at s_j ensures the execution of a path containing s_i, s_j, and s_l in that order. Hence, the p-use must have been covered by the test cases that distinguished the *true* and *false* mutants at s_j.

SDMU (1)



$$
\begin{array}{ll}
 & \bullet \\
 & \bullet \\
s_i: & x = expr \\
 & \bullet \\
 & \bullet \\
s_{j_1}: & (y_1 = f_1(x, \cdots)) \text{ or } pred_1(x, \cdots) \\
 & \bullet \\
 & \bullet \\
s_{j_2}: & (y_2 = f_2(x, \cdots)) \text{ or } pred_2(x, \cdots) \\
 & \bullet \\
 & \bullet \\
s_{j_u}: & (y_u = f_u(x, \cdots)) \text{ or } pred_u(x, \cdots) \\
 & \bullet \\
 & \bullet
\end{array}
$$

Figure 2: Structure of programs with one definition and multiple uses of x. Uses of x in f and $pred$ are, respectively, computational and predicate uses.

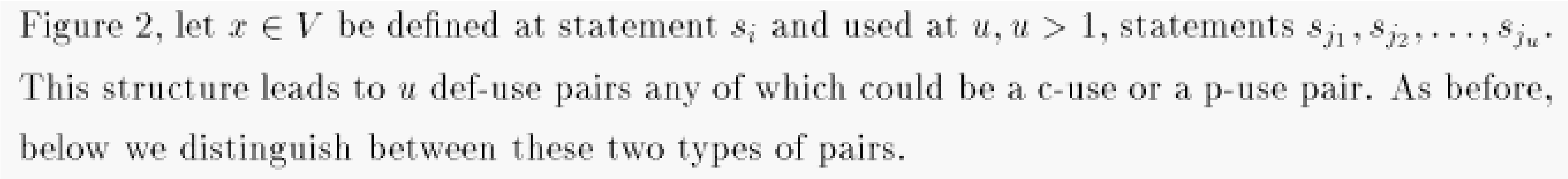
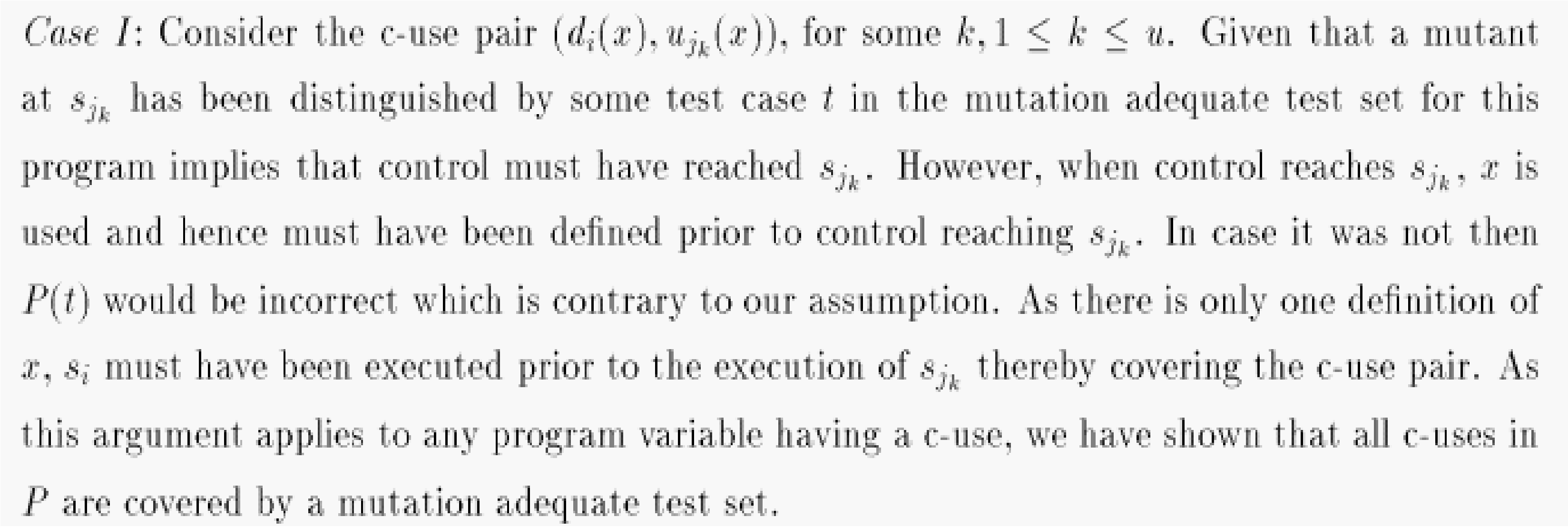


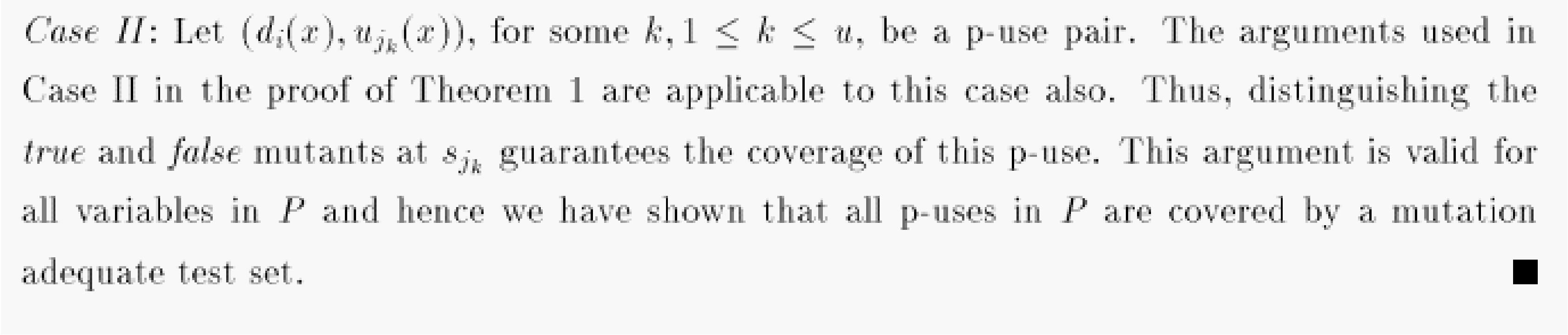
Figure 2, let $x \in V$ be defined at statement s_i and used at $u, u > 1$, statements $s_{j_1}, s_{j_2}, \ldots, s_{j_u}$. This structure leads to u def-use pairs any of which could be a c-use or a p-use pair. As before, below we distinguish between these two types of pairs.

SDMU (2)



Case I: Consider the c-use pair $(d_i(x), u_{j_k}(x))$, for some $k, 1 \leq k \leq u$. Given that a mutant at s_{j_k} has been distinguished by some test case t in the mutation adequate test set for this program implies that control must have reached s_{j_k}. However, when control reaches s_{j_k}, x is used and hence must have been defined prior to control reaching s_{j_k}. In case it was not then $P(t)$ would be incorrect which is contrary to our assumption. As there is only one definition of x, s_i must have been executed prior to the execution of s_{j_k} thereby covering the c-use pair. As this argument applies to any program variable having a c-use, we have shown that all c-uses in P are covered by a mutation adequate test set.

SDMU (3)

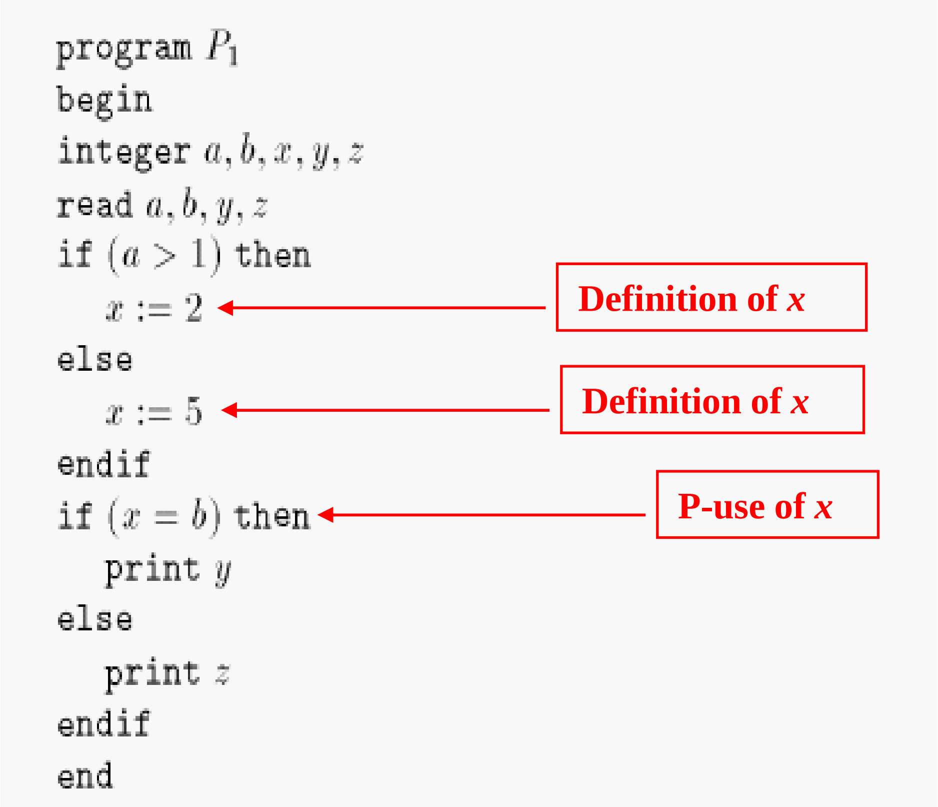


Case II: Let $(d_i(x), u_{j_k}(x))$, for some $k, 1 \leq k \leq u$, be a p-use pair. The arguments used in Case II in the proof of Theorem 1 are applicable to this case also. Thus, distinguishing the *true* and *false* mutants at s_{j_k} guarantees the coverage of this p-use. This argument is valid for all variables in P and hence we have shown that all p-uses in P are covered by a mutation adequate test set. ■

MDSU (1)

- Lemma 1 *For Category III, M_R subsumes CU*
- Proof: The proof follows from the arguments used in Case I of the proof of Theorem 1 applied to all c-use pairs

MDSU (2)



```
program $P_1$
begin
integer $a, b, x, y, z$
read $a, b, y, z$
if $(a > 1)$ then
    $x := 2$
else
    $x := 5$
endif
if $(x = b)$ then
    print $y$
else
    print $z$
endif
end
```

Definition of *x*

Definition of *x*

P-use of *x*

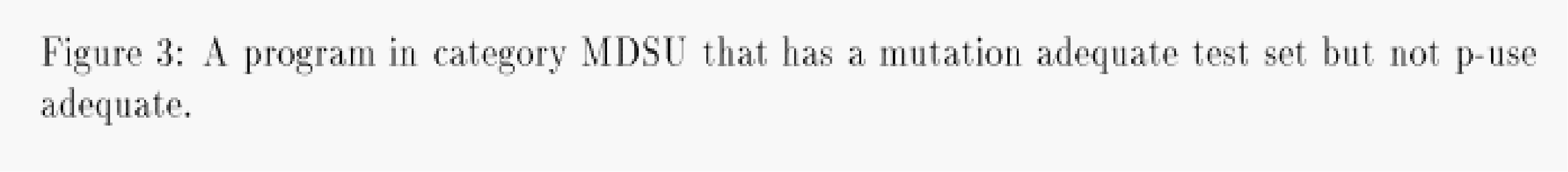


Figure 3: A program in category MDSU that has a mutation adequate test set but not p-use adequate.

MDSU (3)

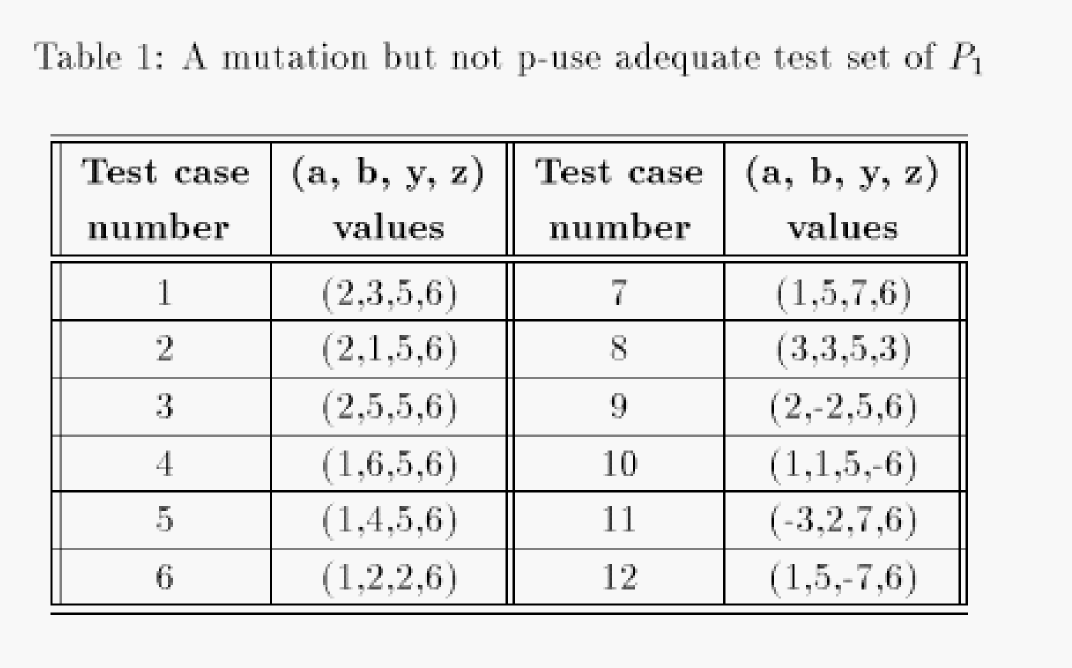


Table 1: A mutation but not p-use adequate test set of P_1

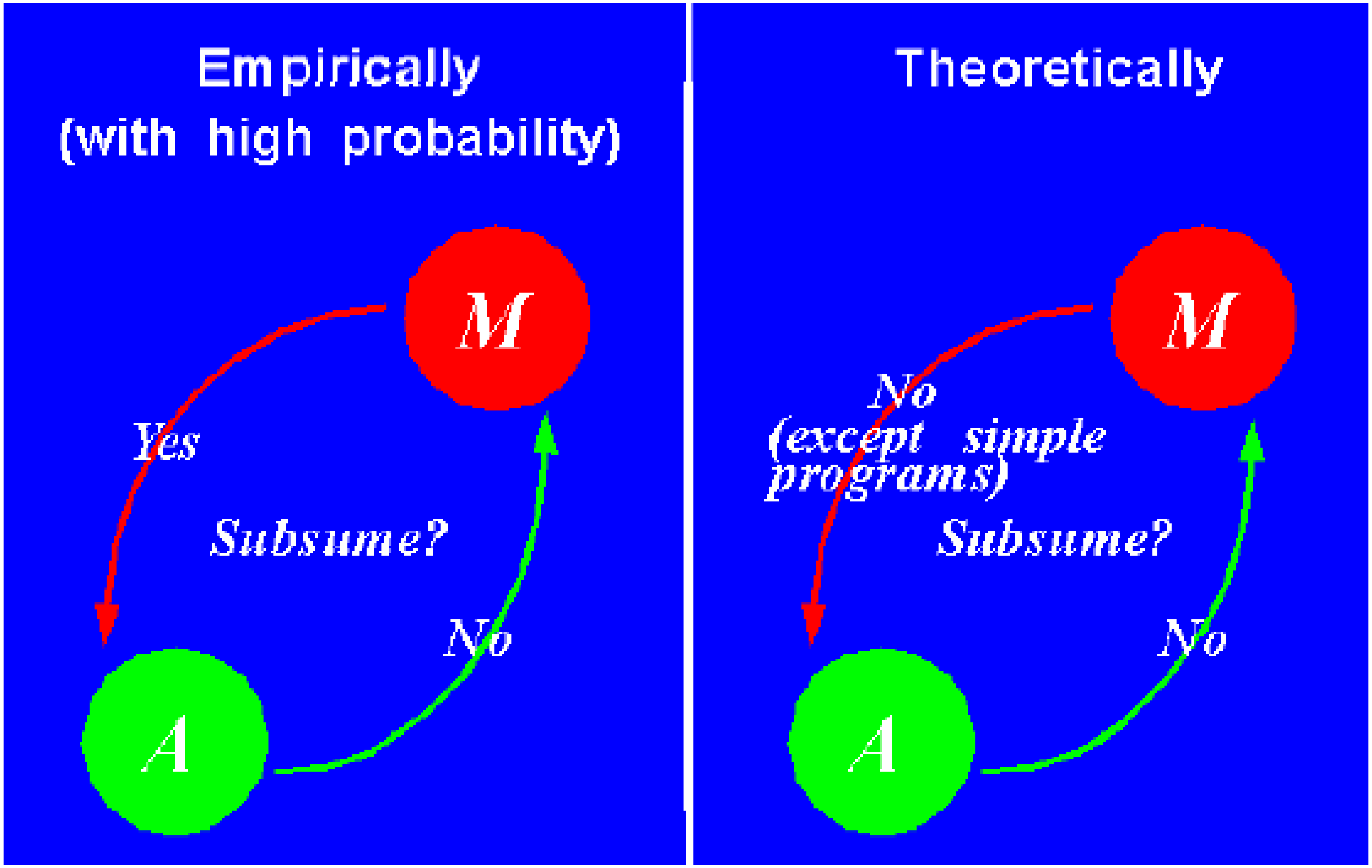
Test case number	(a, b, y, z) values	Test case number	(a, b, y, z) values
1	(2,3,5,6)	7	(1,5,7,6)
2	(2,1,5,6)	8	(3,3,5,3)
3	(2,5,5,6)	9	(2,-2,5,6)
4	(1,6,5,6)	10	(1,1,5,-6)
5	(1,4,5,6)	11	(-3,2,7,6)
6	(1,2,2,6)	12	(1,5,-7,6)

- Proof: Figure 3 shows a program that has two p-use pairs for variable *x*. Table 1 lists a *mutation adequate test set which does not cover the p-use* pair consisting of the definition *x*:=2 and its use in the predicate *x*=*b* because the successor print *y* is not executed.

Empirical Study : Subsumption

- All-uses scores using mutation adequate test sets are, in general, higher than the mutation scores using all-uses adequate test sets.

Conclusion on Subsumption



Cost Metrics

- Number of executions
- Number of test cases
- Test case generation
- Learning testing tools
- Identifying equivalent mutants & infeasible all-uses

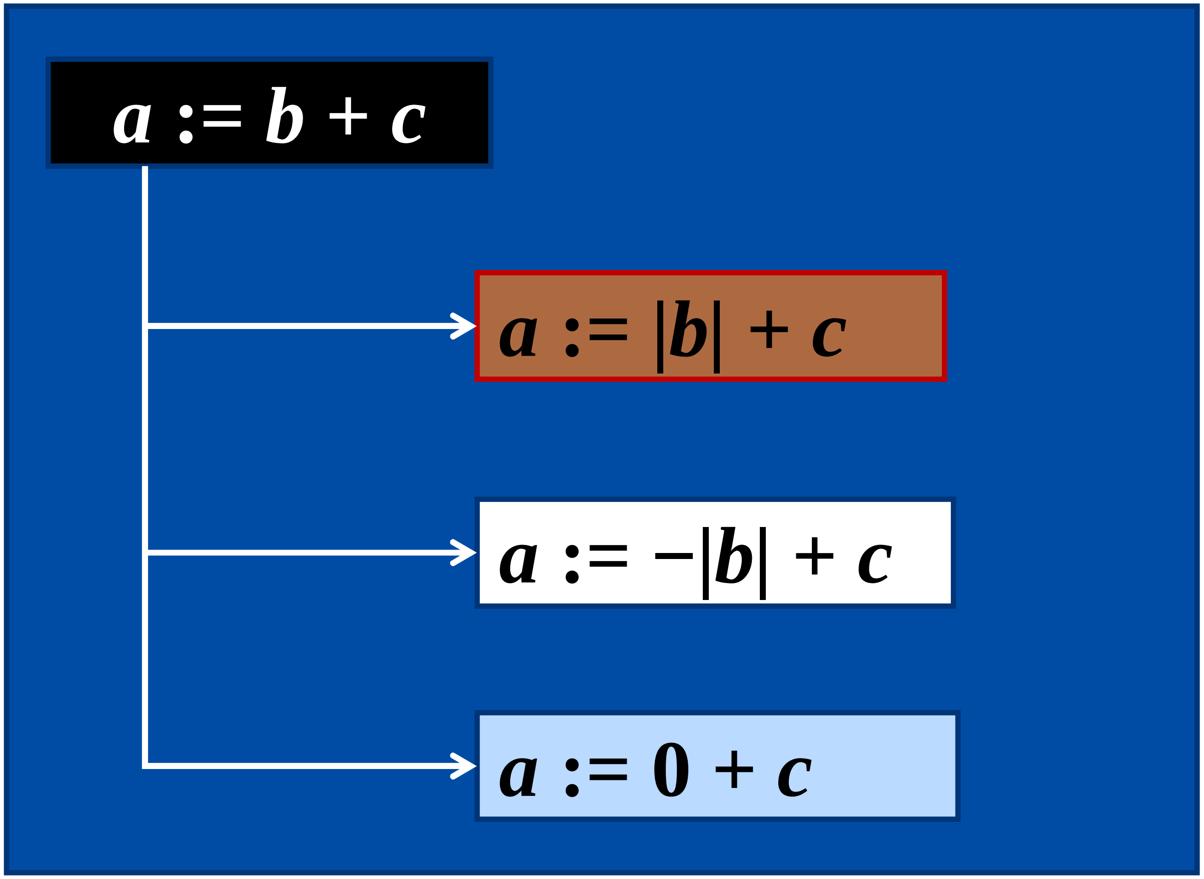
Reducing Mutation Cost

- The cost of mutation testing can be reduced if the number of mutants to be examined is reduced

Selection Mutation

- Select proper mutant operators
  - *ror*: relational operator replacement
  - *lcr*: logical connector replacement
  - *abs*: absolute value insertion
  - *sdl*: statement deletion

abs Mutant Operator



$a := b + c$

$a := |b| + c$

$a := -|b| + c$

$a := 0 + c$

ror Mutant Operator

relational operator replacement

$a < b + c$

>   >=   <=   ==

!=   ***true***   ***false***

Random x% Mutation

- Randomly select a small percentage of mutants from each mutant type

Weak Mutation (1)

- *Reachability*
  - Execute the mutated statement
- *Necessity*
  - Make a state change
- ~~*Sufficiency*~~
  - ~~Propagate the change to output~~

Weak Mutation - Advantage

- Weak mutation *reduces the amount of execution* for distinguishing each mutant

Weak Mutation - Disadvantage

- The disadvantage of weak mutation testing is that there is no guarantee that the different immediate effect will cause a different final result.

Weak Mutation (2)

- *Weak* mutation is as effective as *strong* mutation if the weak mutation hypothesis is true:
  - (reachability and necessity) → sufficiency
- Experiments have shown that this is *true for 61% of all the cases studied*

Weak Mutation (3)

- Suppose that the weak mutation hypothesis does not hold for a particular fault (say *F*). That is, there exists *a non–empty input set* that satisfies the reachability and necessity conditions *while not producing a detectable failure.*

- But in the code under test, there will be many locations with potential faults, each with its own reachability and necessity conditions. It may be that satisfying those other conditions will force the execution of this fault (i.e., *F*) in a way that must produce a detectable failure (namely to satisfy the sufficiency condition).

- Thus, the weak mutation hypothesis may not hold when a single fault is considered alone, but may hold when the fault is considered as part of a larger program (which has many faults).

Reduction Measurement (1)

- Size reduction:

$$1-\frac{\text{Average size of test sets adequate with respect to alternate mutation}}{\text{Average size of mutation adequate test sets}}$$

- Expense reduction:

$$1-\frac{\text{Total number of mutants examined when using alternate mutation}}{\text{Total number of mutants examined in mutation}}$$

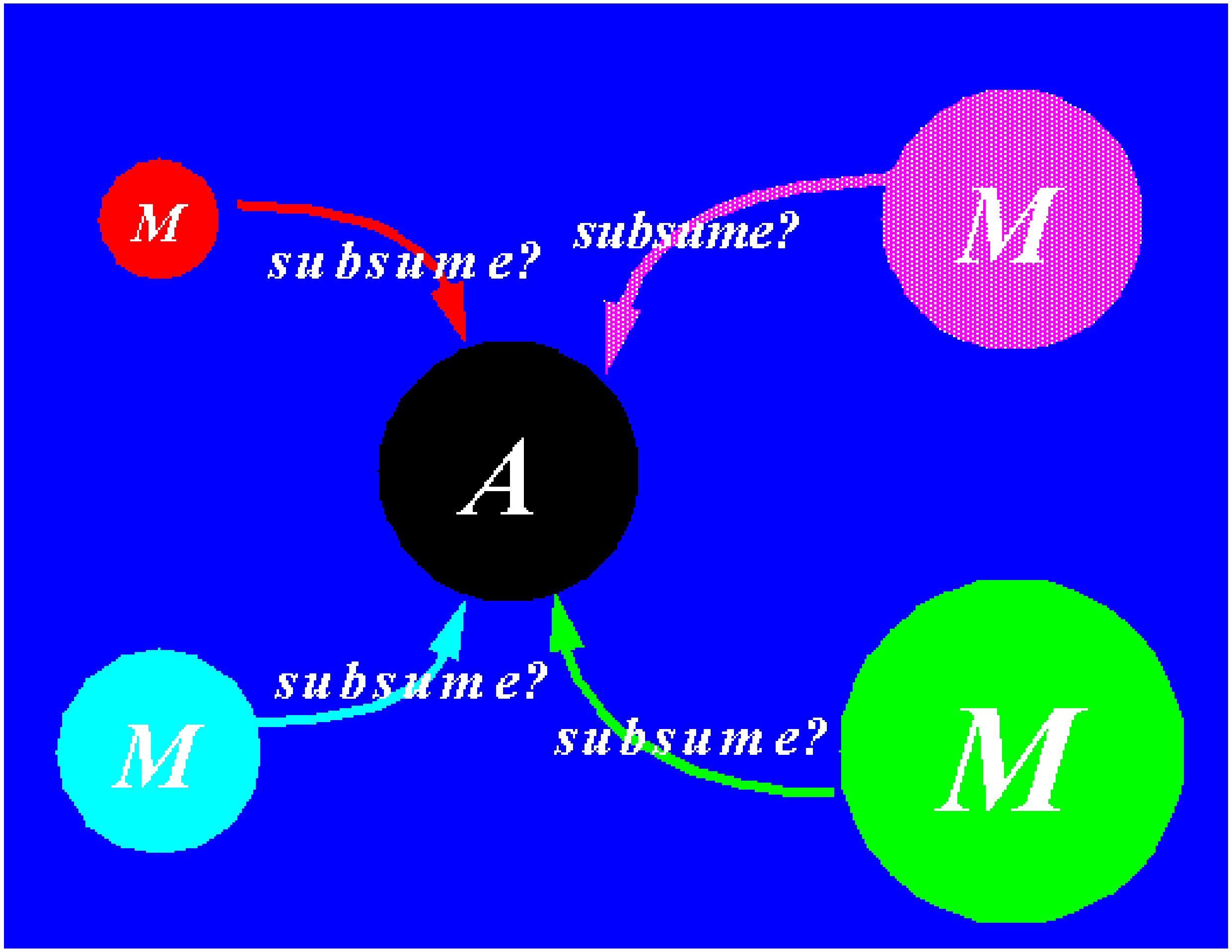
Reduction Measurement (2)

- Mutation score reduction
- All-uses scores reduction

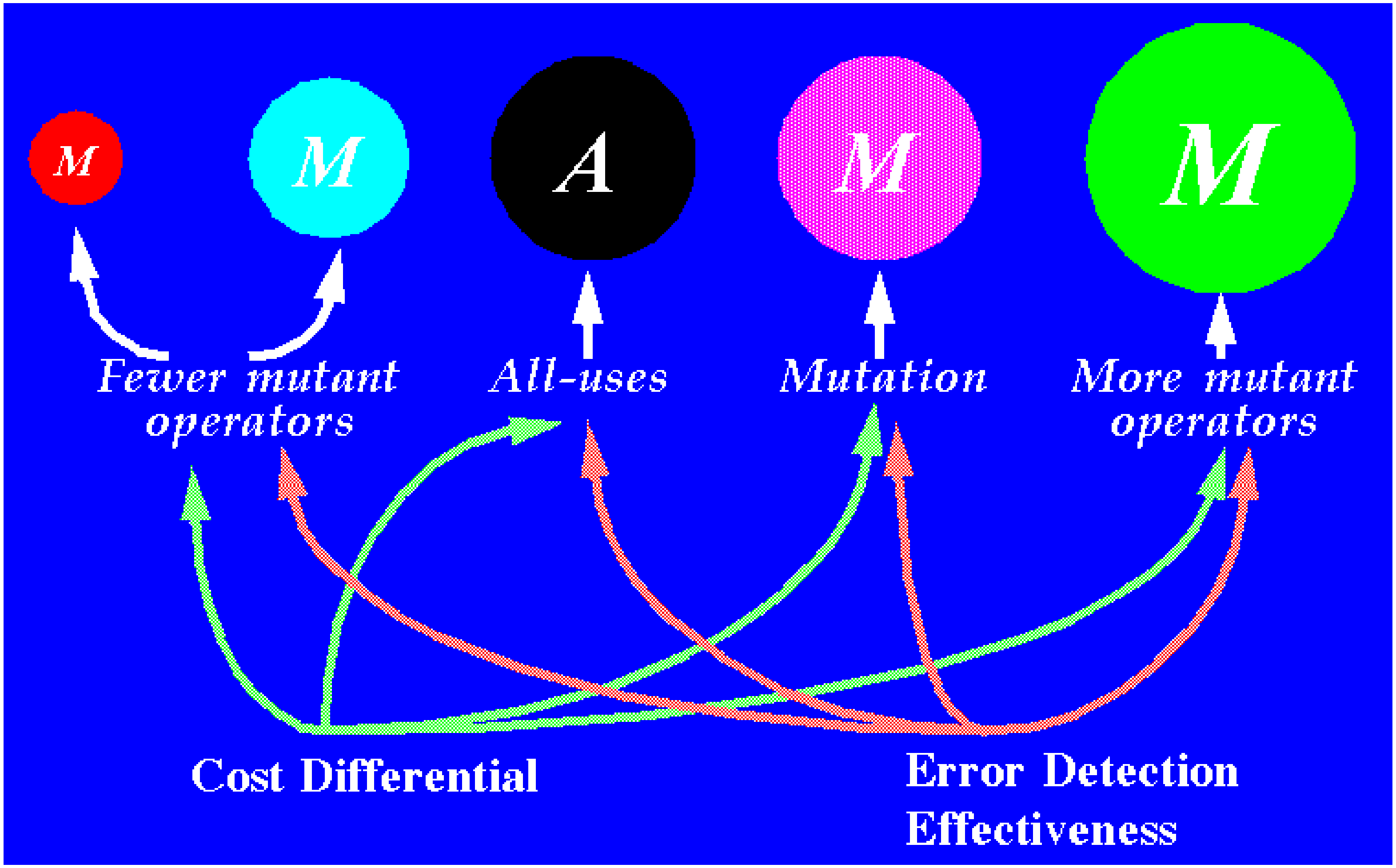
Observation

- Compared to mutation, randomly selected *x*% mutation and *abs*/*ror* mutation provide:
  - Significant size reduction
  - Significant expense reduction
  - Small reduction on mutation scores
  - Small reduction on all-uses scores

Alternate Mutation



Incremental Testing Strategy


M
M
A
M
M
Fewer mutant operators
All-uses
Mutation
More mutant operators
Cost Differential
Error Detection Effectiveness