Mutation testing: practical aspects and cost analysis

Macario Polo and Mario Piattini
Alarcos Group
Department of Information Systems and Technologies
University of Castilla-La Mancha (Spain)

Macario.Polo@uclm.es

Contents

- What is mutation testing?
- Mutation operators
- Steps of mutation testing
- Cost reduction
- Conclusions

What is mutation testing?

- The goal of testing is to find faults on the System Under Test (SUT)
- Thus, a test suite is more or less effective depending on its ability to find faults on the SUT

A possible “SUT”

To be, or not to be: that is de question:
Whether 'tis novler in the mind to sufer
The slings and arrows of outrageus fortune,
Or to take arms against a sea of troubles,
And by opposing end them? To die: to sleep;
No more; and by a slept to say the end
The heart-ache and the thousand natural socks
That flesh is heir to, 'tis a consummation
Deboutly to be wish'd. To die, to sleep; ...
(Hamlet, by William Shakespeare)

What is mutation testing?

- If I need to select an English reviewer, maybe I would select that of you who more faults have found
- Mutation works in this way:
  - A set of “mutants” are generated from a given SUT
  - A test suite is executed against the original SUT and its mutants
  - The adequacy of the suite is measured in terms of the “mutation score”

What is mutation testing?

- The “mutation score” measures the ability of the test suite to find faults on the SUT

$$MS(P,T) = \frac{K}{(M - E)}$$

- …where:
  - *P*: program under test
  - *T*: test suite
  - *K*: number of mutants “killed”
  - *M*: number of mutants
  - *E*: number of functionally-equivalent mutants

What is mutation testing?

- Mutants maybe “killed” or “alive”
  - A mutant *m* is “killed” when it shows a different behavior that *P*, for one or more $t \in P$:

$$\exists \mathrm{t} \in \mathrm{T} \ / \ f(m,t) \neq f(P,t)$$

  - Otherwise, the mutant is “alive”

$$f(m,t) = f(P,t) \quad \forall \mathrm{t} \in \mathrm{T}$$

What is mutation testing?

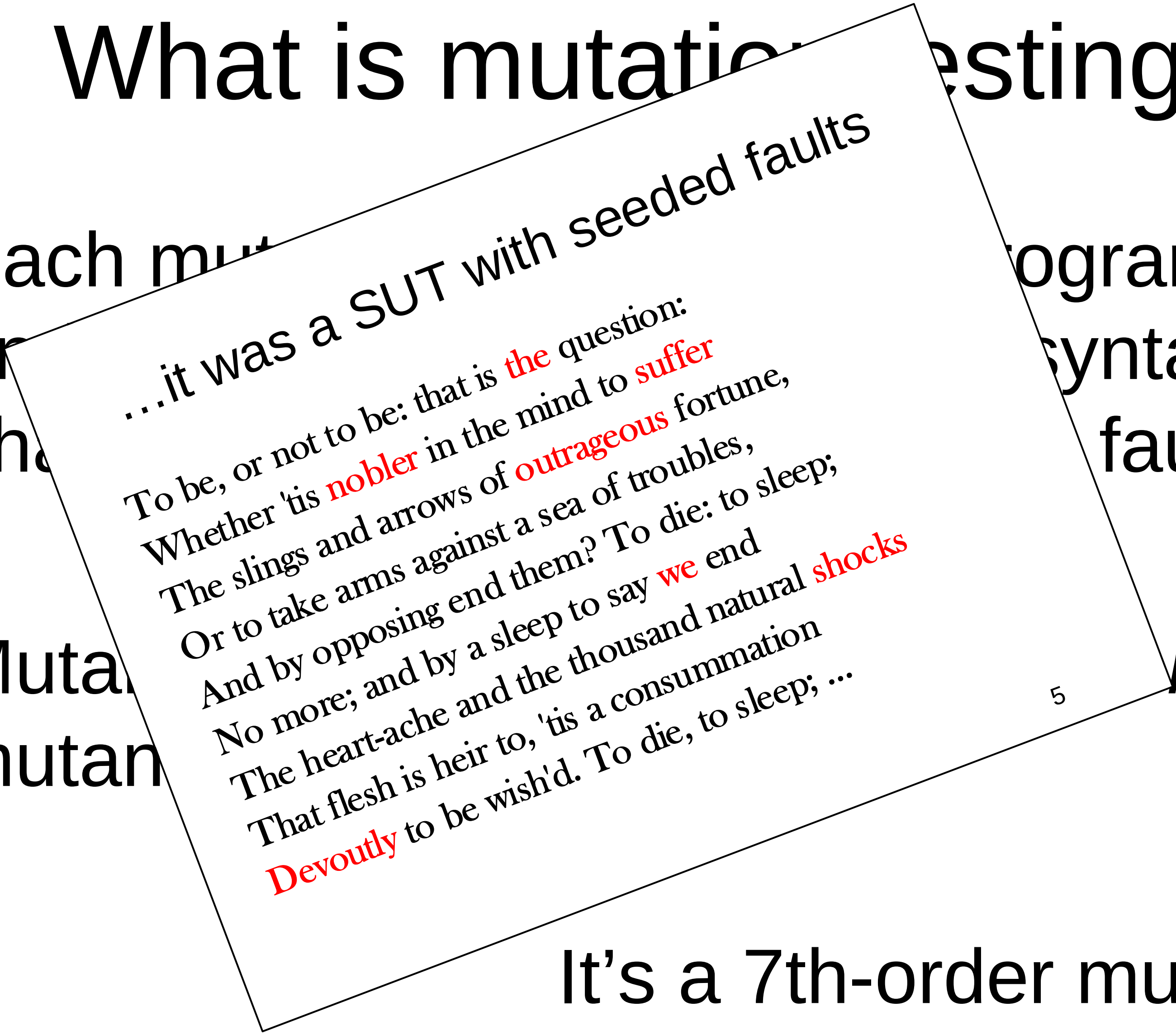
- Each mutant is a copy of the program under test, usually with a small syntactic change, which is interpreted as a fault

- Mutants with *n* faults are call *n-order* mutants

What is mutati esting?

- Each mu ogram
  u syntactic
  ch fault

- Muta r
  mutan



…it was a SUT with seeded faults

To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune,
Or to take arms against a sea of troubles,
And by opposing end them? To die: to sleep;
No more; and by a sleep to say we end
The heart-ache and the thousand natural shocks
That flesh is heir to, 'tis a consummation
Devoutly to be wish'd. To die, to sleep; ...

5

It’s a 7th-order mutant

A program and four mutants

Version	**Code**

P (original)

```
int sum(int a, int b) {
   return a + b;
}
```

Mutant 1

```
int sum(int a, int b) {
   return a - b;
}            ↑
```

Mutant 2

```
int sum(int a, int b) {
   return a * b;
}            ↑
```

Mutant 3

```
int sum(int a, int b) {
   return a / b;
}            ↑
```

Mutant 4

```
int sum(int a, int b) {
   return a + b++;
}               ↑
```

	Test data (a,b)			
	(1, 1)	**(0, 0)**	**(-1, 0)**	**(-1, -1)**
P	2	0	-1	-2
M1	0	0	-1	0
M2	1	0	0	1
M3	1	Error	Error	1
M4	2	0	-1	-2

What is mutation testing?

- Faults are introduced (“seeded”) by mutation operators, and try to imitate actual programmer errors
- Mutation relies on the “Coupling effect”: a test suite that detects all simple faults in a program is so sensitive that it also detects more complex faults

Some mutation operators

Operator	**Description**
ABS	Substitution of a variable *x* by *abs(x)*
ACR	Substitution of a variable array reference by a constant
AOR	Arithmetic operator replacement (*a+b* by *a-b)*
CRP	Substitution of a constant value
ROR	Relational operator replacement (*A and B* by *A or B*)
RSR	Return statement substitution (*return 5* by *return 0*)
SDL	Removal of a sentence
UOI	Unary operator insertion (instead of *x*, write –*x*)
…	…

Steps of mutation testing

- Mutation testing has three main steps:
  1. Mutant generation
  2. Mutant execution
  3. Result analysis

Mutant generation

- Almost each executable instruction of the original program can be mutated with several mutation operators
- Therefore, the number of mutants generated for a normal program may be huge
- The cost of compilation of all mutants may be also significant

Mutant generation

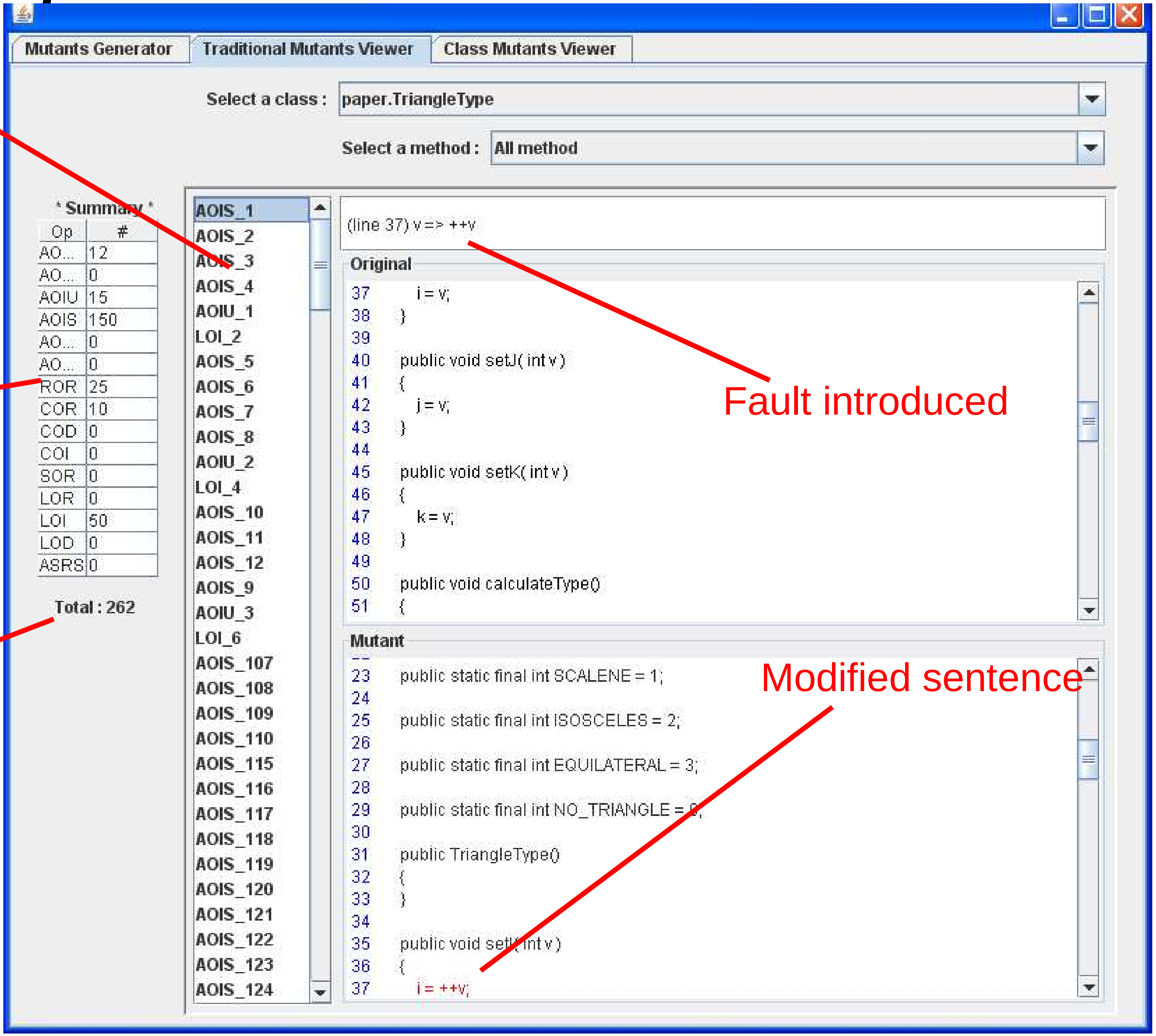
- Offutt et al. (1996): 10 programs from 10 to 48 executable sentences produce from 183 to 3010 mutants
- Mresa and Botaci (1999): 11 programs with 43,7 LOC produce 3211 mutants
- One of our experiments: mean of 76,7 LOC produce a mean of 150 mutants

Mutant generation

- Java version of Myers (1979) triangle-type problem: 61 LOC, 262 mutants

- A widely-used mutation tool is MuJava (Ma, Offutt and Kwon, 2005)

Mutant generation: the MuJava tool



Select a class : paper.TriangleType

Select a method : All method

* Summary *

Op	#
AO...	12
AO...	0
AOIU	15
AOIS	150
AO...	0
AO...	0
ROR	25
COR	10
COD	0
COI	0
SOR	0
LOR	0
LOI	50
LOD	0
ASRS	0

Total : 262

- AOIS_1
- AOIS_2
- AOIS_3
- AOIS_4
- AOIU_1
- LOI_2
- AOIS_5
- AOIS_6
- AOIS_7
- AOIS_8
- AOIU_2
- LOI_4
- AOIS_10
- AOIS_11
- AOIS_12
- AOIS_9
- AOIU_3
- LOI_6
- AOIS_107
- AOIS_108
- AOIS_109
- AOIS_110
- AOIS_115
- AOIS_116
- AOIS_117
- AOIS_118
- AOIS_119
- AOIS_120
- AOIS_121
- AOIS_122
- AOIS_123
- AOIS_124

(line 37) v => ++v

Original

```
37        i = v;
38     }
39
40     public void setJ( int v )
41     {
42        j = v;
43     }
44
45     public void setK( int v )
46     {
47        k = v;
48     }
49
50     public void calculateType()
51     {
```

Mutant

```
23     public static final int SCALENE = 1;
24
25     public static final int ISOSCELES = 2;
26
27     public static final int EQUILATERAL = 3;
28
29     public static final int NO_TRIANGLE = 0;
30
31     public TriangleType()
32     {
33     }
34
35     public void setI( int v )
36     {
37        i = ++v;
```

Mutants (and operators)

Number of mutants per operator

Number of mutants

Fault introduced

Modified sentence

Mutant generation: the MuJava tool

- In general, a parser is required to generate mutants:
  - *a+b* is translated into *a-b, a*b, a/b*
  - Then, these program versions are compiled
- MuJava uses “Mutant Schemata Generation”
  - With some operators, it substitutes (at bytecode level) *a+b* by *a OPERATOR b*
  - Then, all the program versions are directly generated with no need of compiling

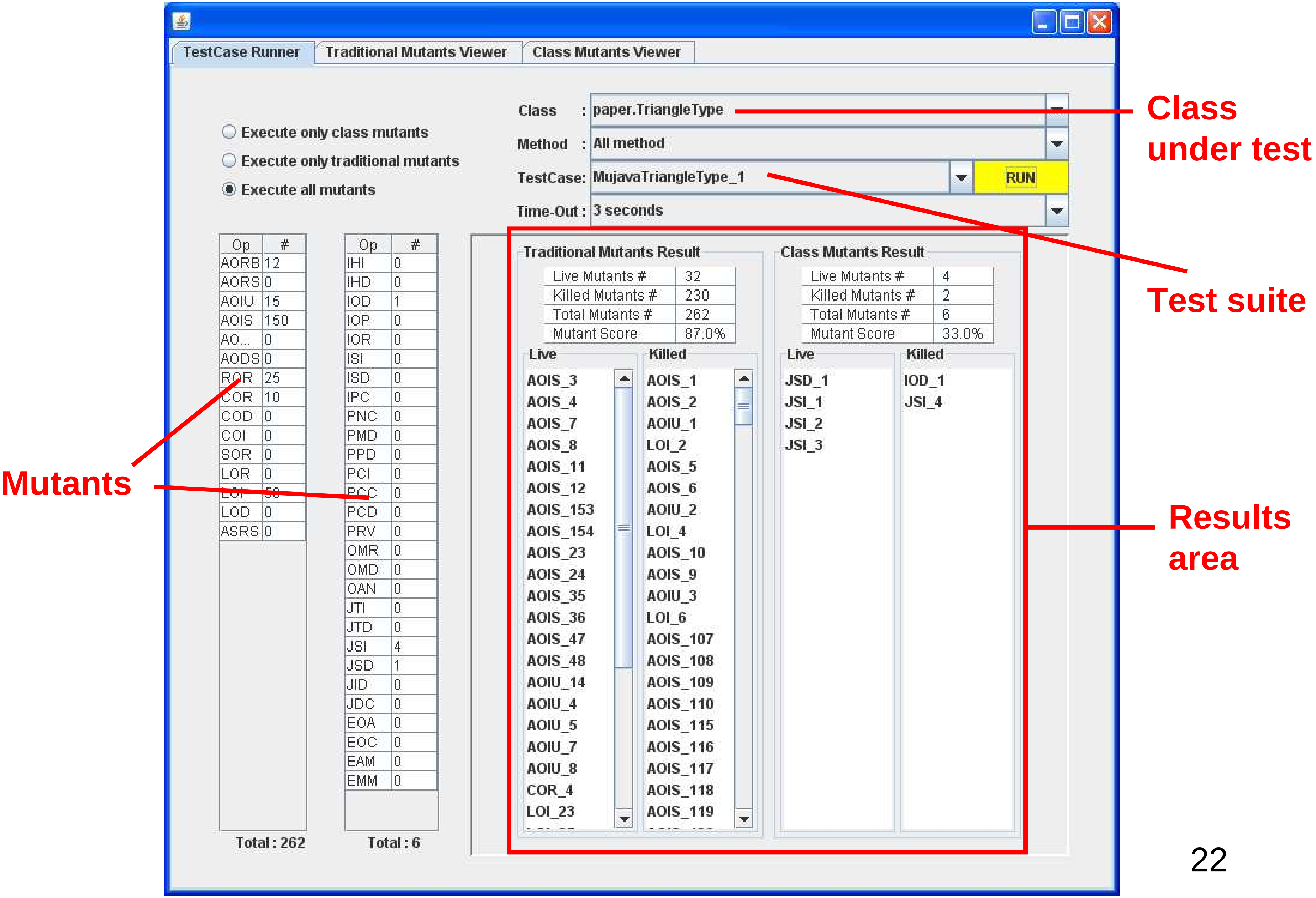
Mutant execution

- In this case, the problem is the huge number of test cases that must be executed: each case is executed against the original program and the mutants.
- For testing a simple *BankingAccount* class, with 96 mutants and 300 test cases, 96*300=28,800 executions are required (with at least 28.800 accesses to the database, etc.)

Mutant execution

- All the outputs must be compared to detect which mutants are killed:
  - In the *BankingAccount* example, the outputs of the 300 test cases with the original and the 96 mutants
  - Actually, killed mutants can be removed for further comparisons

Mutant execution: MuJava



○ Execute only class mutants
○ Execute only traditional mutants
● Execute all mutants

Class : paper.TriangleType
Method : All method
TestCase: MujavaTriangleType_1
Time-Out : 3 seconds

Op	#
AORB	12
AORS	0
AOIU	15
AOIS	150
AO...	0
AODS	0
ROR	25
COR	10
COD	0
COI	0
SOR	0
LOR	0
LOI	50
LOD	0
ASRS	0

Total : 262

Op	#
IHI	0
IHD	0
IOD	1
IOP	0
IOR	0
ISI	0
ISD	0
IPC	0
PNC	0
PMD	0
PPD	0
PCI	0
PCC	0
PCD	0
PRV	0
OMR	0
OMD	0
OAN	0
JTI	0
JTD	0
JSI	4
JSD	1
JID	0
JDC	0
EOA	0
EOC	0
EAM	0
EMM	0

Total : 6

Traditional Mutants Result

Live Mutants #	32
Killed Mutants #	230
Total Mutants #	262
Mutant Score	87.0%

Live	Killed
AOIS_3	AOIS_1
AOIS_4	AOIS_2
AOIS_7	AOIU_1
AOIS_8	LOI_2
AOIS_11	AOIS_5
AOIS_12	AOIS_6
AOIS_153	AOIU_2
AOIS_154	LOI_4
AOIS_23	AOIS_10
AOIS_24	AOIS_9
AOIS_35	AOIU_3
AOIS_36	LOI_6
AOIS_47	AOIS_107
AOIS_48	AOIS_108
AOIU_14	AOIS_109
AOIU_4	AOIS_110
AOIU_5	AOIS_115
AOIU_7	AOIS_116
AOIU_8	AOIS_117
COR_4	AOIS_118
LOI_23	AOIS_119

Class Mutants Result

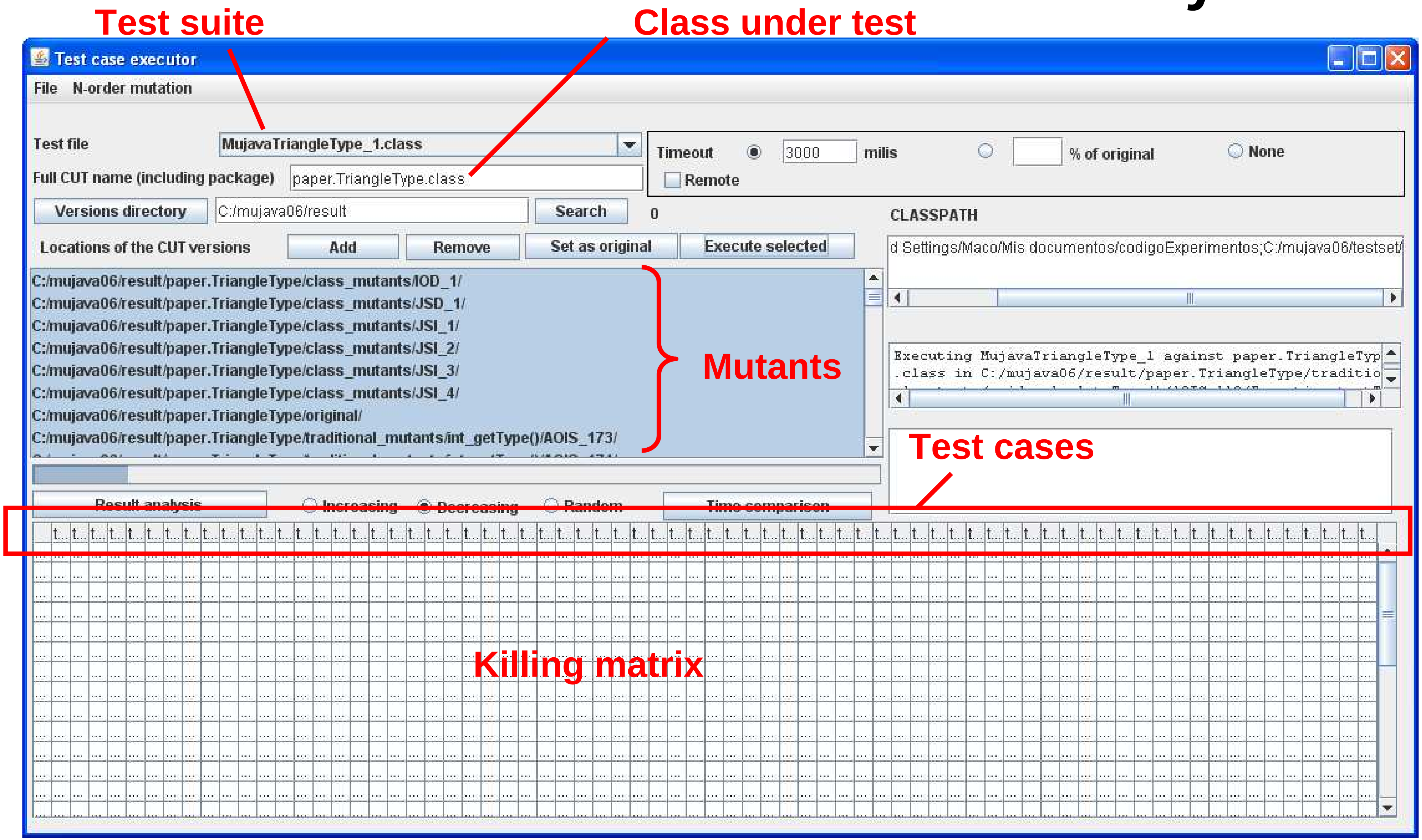
Live Mutants #	4
Killed Mutants #	2
Total Mutants #	6
Mutant Score	33.0%

Live	Killed
JSD_1	IOD_1
JSI_1	JSI_4
JSI_2	
JSI_3	

Mutant execution: *testooj*

- *testooj* is a relatively user-friendly research tool developed in the University of Castilla-La Mancha
- Generates test cases in several formats and according to several generation strategies
- Executes test cases against versions and gives some additional results

Mutant execution: *testooj*



Test suite

Class under test

Mutants

Test cases

Killing matrix

Test file: MujavaTriangleType_1.class

Full CUT name (including package): paper.TriangleType.class

Timeout ● 3000 milis ○ % of original ○ None

☐ Remote

Versions directory: C:/mujava06/result

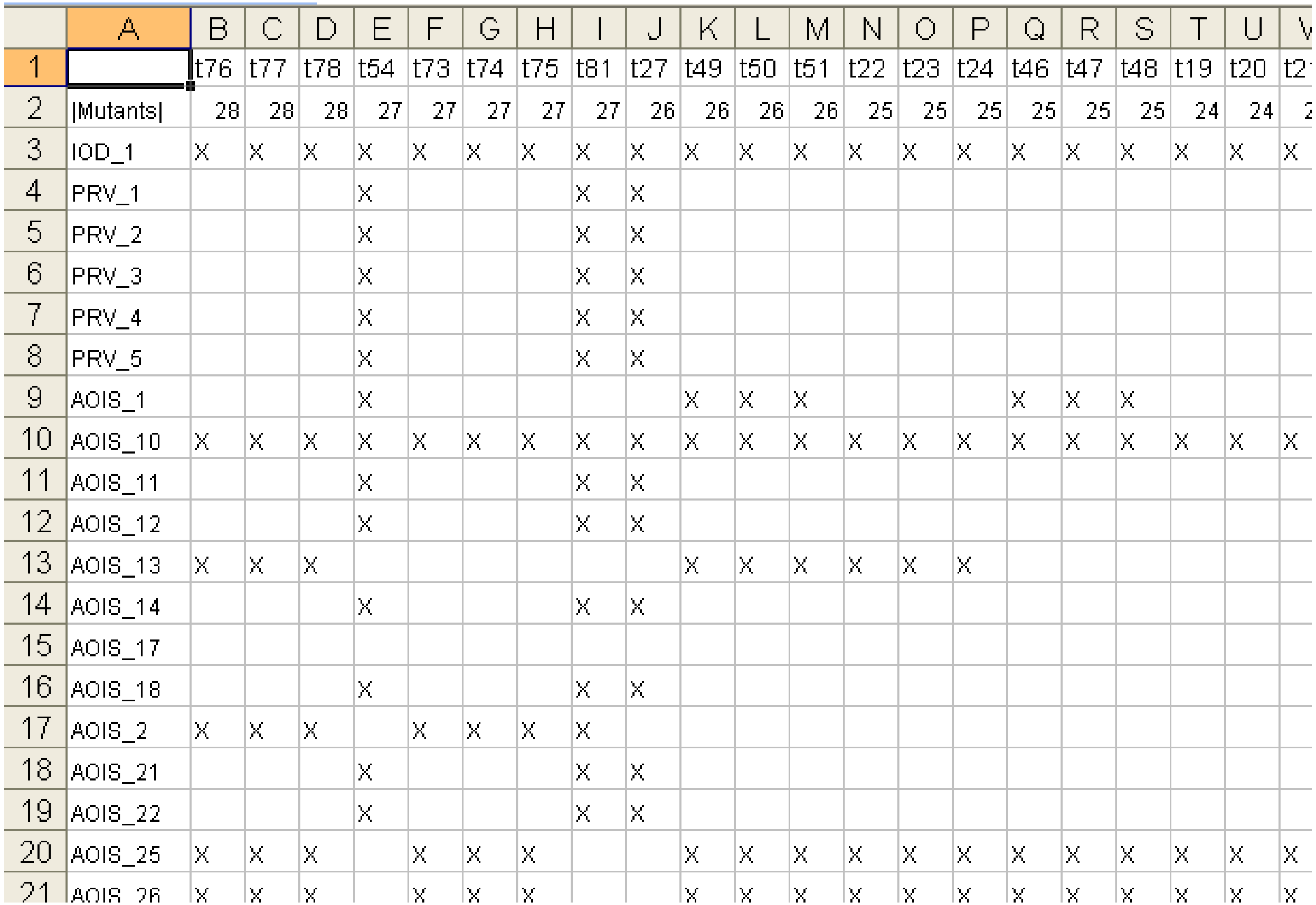
0

C:/mujava06/result/paper.TriangleType/class_mutants/IOD_1/
C:/mujava06/result/paper.TriangleType/class_mutants/JSD_1/
C:/mujava06/result/paper.TriangleType/class_mutants/JSI_1/
C:/mujava06/result/paper.TriangleType/class_mutants/JSI_2/
C:/mujava06/result/paper.TriangleType/class_mutants/JSI_3/
C:/mujava06/result/paper.TriangleType/class_mutants/JSI_4/
C:/mujava06/result/paper.TriangleType/original/
C:/mujava06/result/paper.TriangleType/traditional_mutants/int_getType()/AOIS_173/

CLASSPATH: d Settings/Maco/Mis documentos/codigoExperimentos;C:/mujava06/testset/

```
Executing MujavaTriangleType_1 against paper.TriangleTyp
.class in C:/mujava06/result/paper.TriangleType/traditio
```

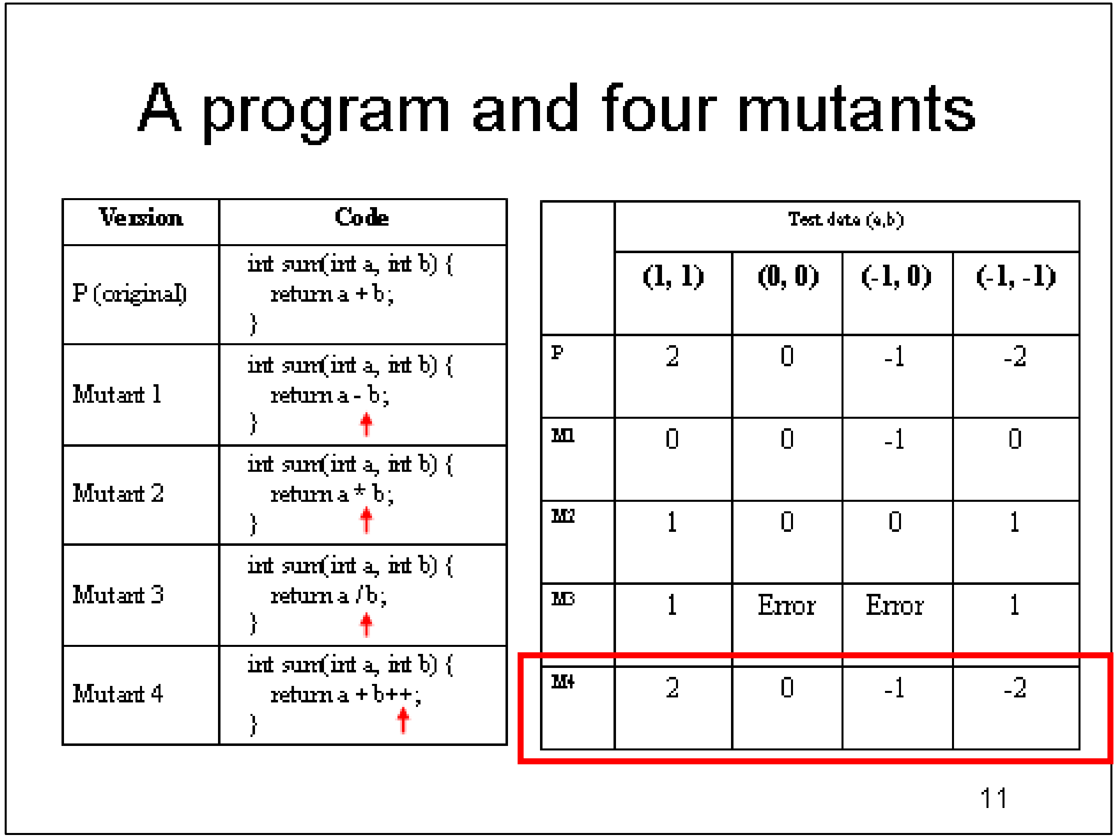
The killing matrix



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
1		t76	t77	t78	t54	t73	t74	t75	t81	t27	t49	t50	t51	t22	t23	t24	t46	t47	t48	t19	t20	t2
2	|Mutants|	28	28	28	27	27	27	27	27	26	26	26	26	25	25	25	25	25	25	24	24	2
3	IOD_1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
4	PRV_1				X				X	X												
5	PRV_2				X				X	X												
6	PRV_3				X				X	X												
7	PRV_4				X				X	X												
8	PRV_5				X				X	X												
9	AOIS_1				X						X	X	X				X	X	X			
10	AOIS_10	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
11	AOIS_11				X				X	X												
12	AOIS_12				X				X	X												
13	AOIS_13	X	X	X							X	X	X	X	X	X						
14	AOIS_14				X				X	X												
15	AOIS_17																					
16	AOIS_18				X				X	X												
17	AOIS_2	X	X	X		X	X	X	X													
18	AOIS_21				X				X	X												
19	AOIS_22				X				X	X												
20	AOIS_25	X	X	X		X	X	X			X	X	X	X	X	X	X	X	X	X	X	X
21	AOIS_26	X	X	X		X	X	X			X	X	X	X	X	X	X	X	X	X	X	X

Result analysis

- The major difficulties appear with the detection of functionally equivalent mutants



A program and four mutants

Version: P (original)

```
int sum(int a, int b) {
  return a + b;
}
```

Version: Mutant 1

```
int sum(int a, int b) {
  return a - b;
}
```

Version: Mutant 2

```
int sum(int a, int b) {
  return a * b;
}
```

Version: Mutant 3

```
int sum(int a, int b) {
  return a /b;
}
```

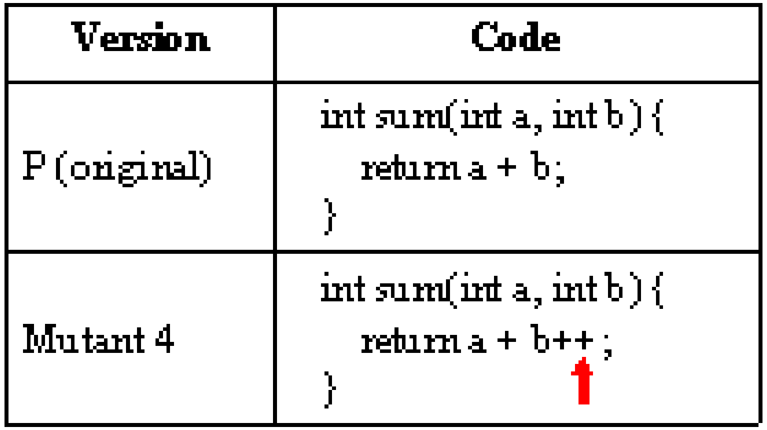
Version: Mutant 4

```
int sum(int a, int b) {
  return a + b++;
}
```

	Test data (a,b)			
	(1, 1)	**(0, 0)**	**(-1, 0)**	**(-1, -1)**
P	2	0	-1	-2
M1	0	0	-1	0
M2	1	0	0	1
M3	1	Error	Error	1
M4	2	0	-1	-2

Result analysis

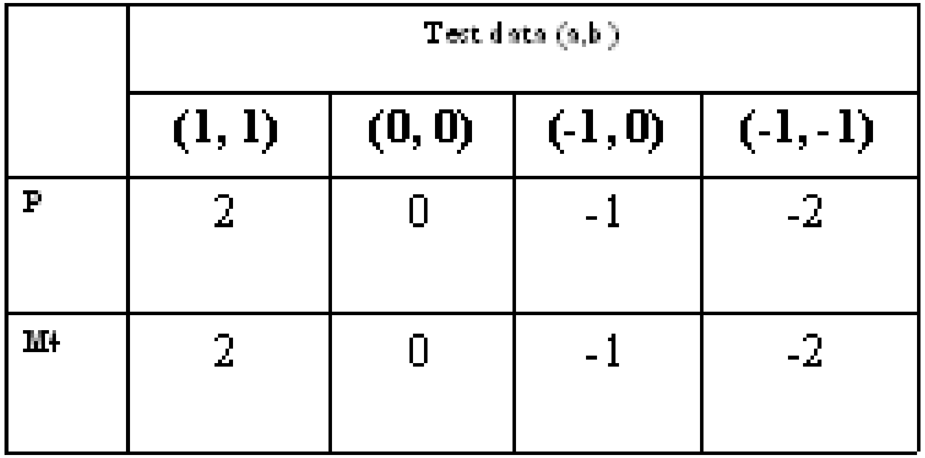
- A functionally equivalent mutant is a mutant which never will be killed
- Actually, the “fault” introduced is not a fault, but a code de-optimization



Version	Code

P (original)	`int sum(int a, int b) {`
	`return a + b;`
	`}`

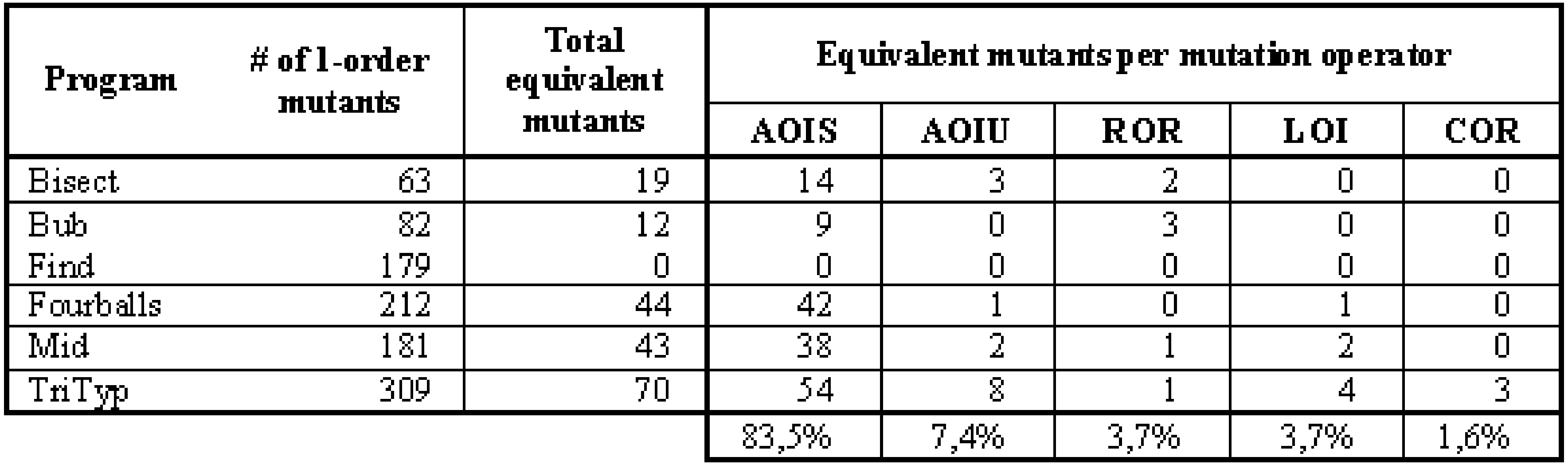
Mutant 4	`int sum(int a, int b) {`
	`return a + b++ ;`
	`}`



	Test data (a,b)			
	(1, 1)	**(0, 0)**	**(-1, 0)**	**(-1, -1)**
P	2	0	-1	-2
M4	2	0	-1	-2

Result analysis

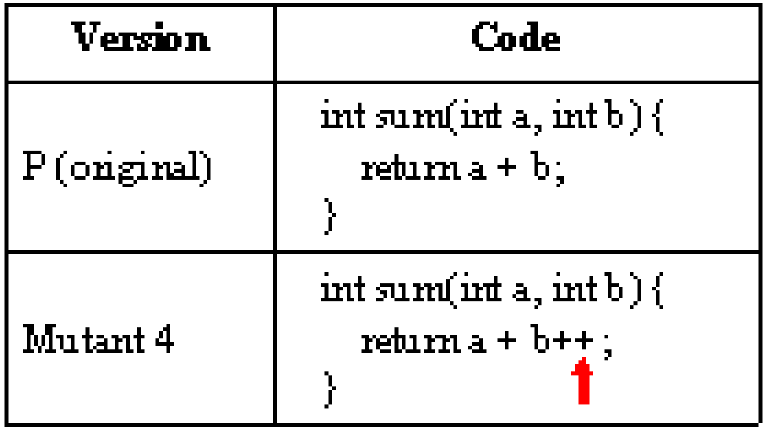
- Mutation operators have different proneness for producing equivalent mutants



Program	# of 1-order mutants	Total equivalent mutants	Equivalent mutants per mutation operator				
			AOIS	AOIU	ROR	LOI	COR
Bisect	63	19	14	3	2	0	0
Bub	82	12	9	0	3	0	0
Find	179	0	0	0	0	0	0
Fourballs	212	44	42	1	0	1	0
Mid	181	43	38	2	1	2	0
TriTyp	309	70	54	8	1	4	3
			83,5%	7,4%	3,7%	3,7%	1,6%

Result analysis

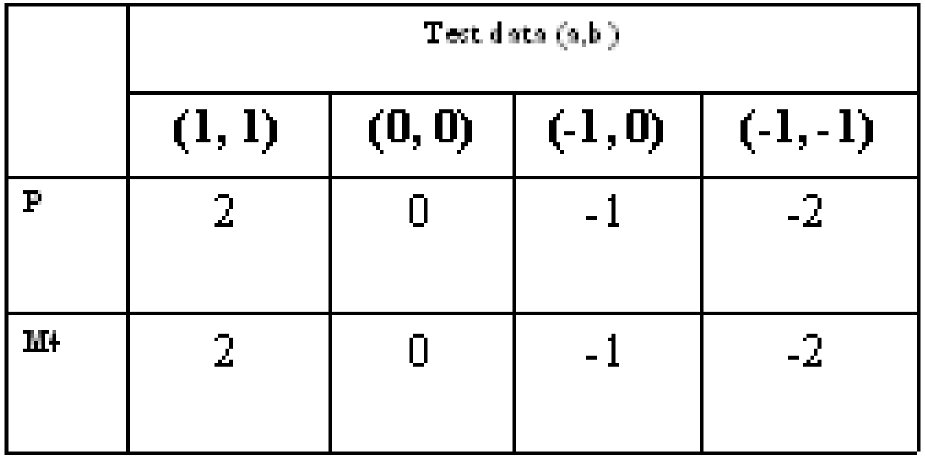
- The example is an occurrence of the AOIS operator



Version	Code

P (original)	int sum(int a, int b) {
	return a + b;
	}

Mutant 4	int sum(int a, int b) {
	return a + b++ ;
	}



	Test data (a,b)			
	(1, 1)	(0, 0)	(-1, 0)	(-1, -1)
P	2	0	-1	-2
M4	2	0	-1	-2

Result analysis

- Detection of equivalent mutants is usually manual
- Grüen, Schuler and Zeller (2009) report that some equivalent mutants require up to 15 minutes to be detected
- Offutt and Pan (1997) demonstrated that is possible to automatically detect almost 50% of functionally equivalent mutants if the program under test is annotated with constraints

Result analysis

- Once more, the selection of the best mutation operators is essential (this is *selective mutation)*:
  - Replacement of numerical constants
  - Negate jump conditions
  - Replacement of arithmetic operators
  - Omission of method calls (instead of *x=foo()*, write *x=0.0*)

Result analysis

- Other strategies rely on *weak mutation:*
  - “Strong” mutation has three conditions:
    - Reachability (the instruction must be reached)
    - Necessity (once the sentences has been reached, the test case must cause an erroneous state on the mutant)
    - Sufficiency (the erroneous state must be propagated to the output)
  - Instead of observing the output of each test case, the idea of *weak mutation* is to detect changes in intermediate states (reachability+necessity)

Cost reduction

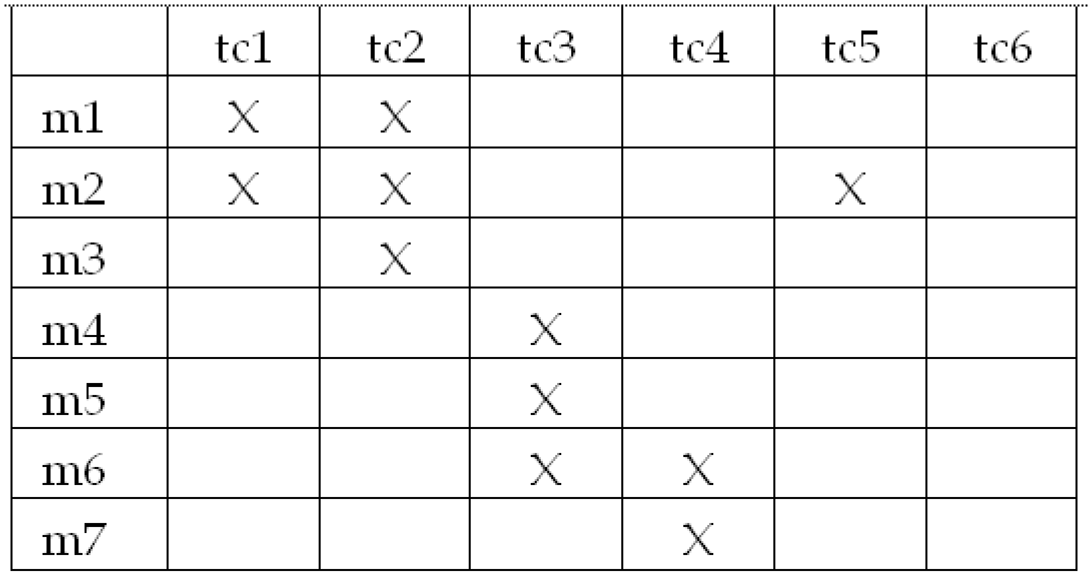
- Summarizing, in mutant generation:
  - Selective mutation (use of the best operators)
  - Mutant Schemata Generation/Mutation at bytecode level
- In mutant execution:
  - Reduction of the test suite
  - N-order mutation
- In result analysis:
  - To take advantage of the previous techniques

Reduction of the test suite

- The optimal test-suite reduction problem:
  - *Given:* Test Suite *T,* a set of test-case requirements *r1, r2, ..., rn,* that must be satisfied to provide the desired test coverage of the program.
  - *Problem:* Find $T' \subset T$ such that *T'* satisfies all *ri* and $(\forall T'' \subset T,\ T''$ satisfies all $r \Rightarrow |T'| \leq |T''|)$
- It is NP-hard (no solution in polynomial time): solutions approached with greedy algorithms

Reduction of the test suite

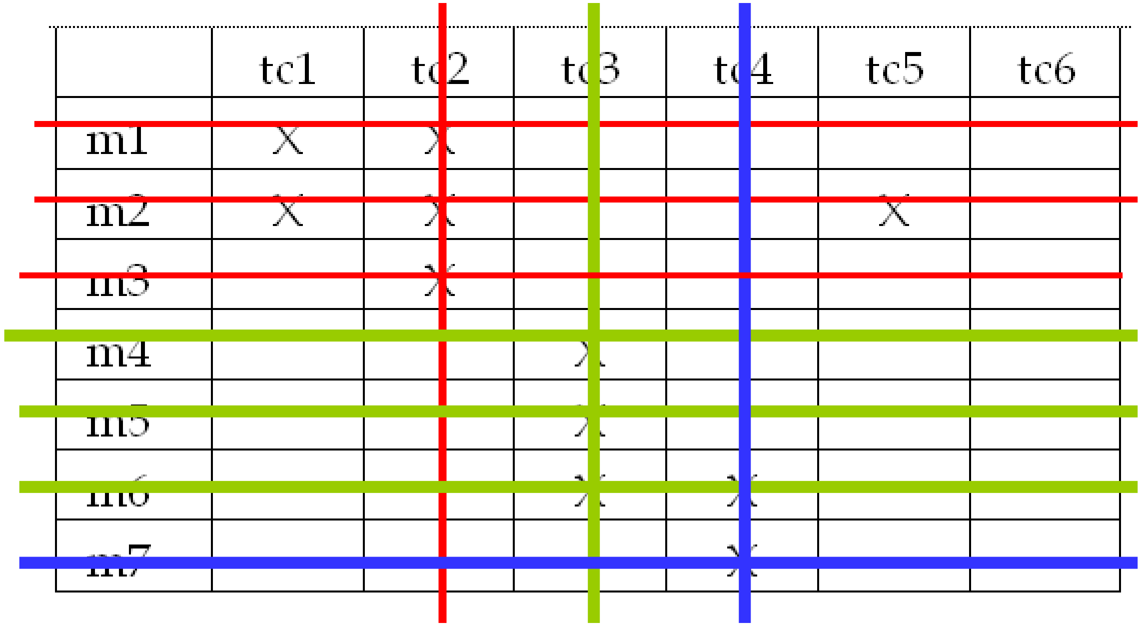
- Example: 6 test cases, 7 mutants



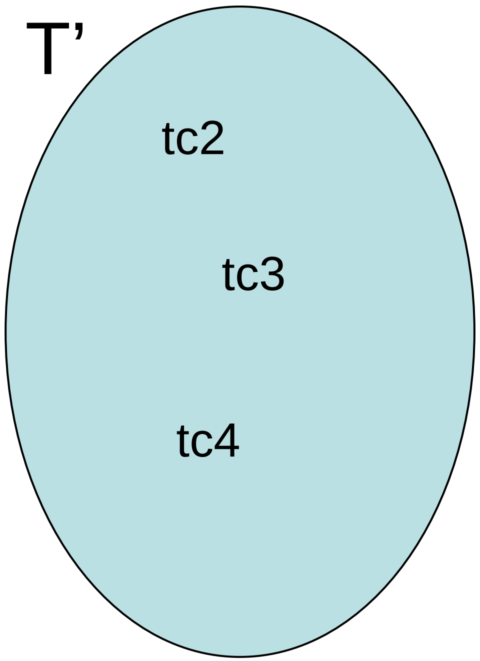
	tc1	tc2	tc3	tc4	tc5	tc6
m1	X	X				
m2	X	X			X	
m3		X				
m4			X			
m5			X			
m6			X	X		
m7				X		

Reduction of the test suite

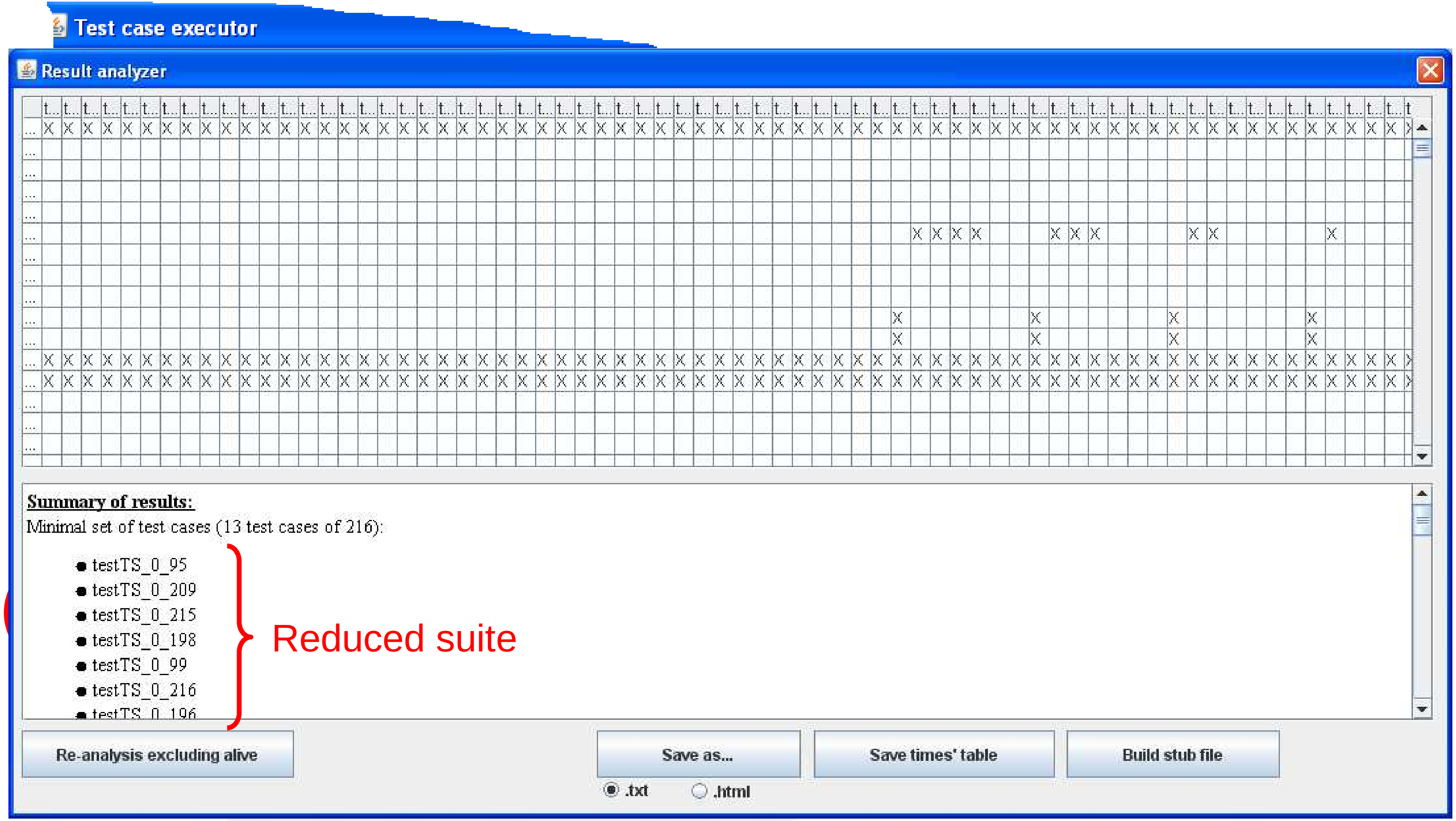
- The greedy approach selects the test case killing more mutants



	tc1	tc2	tc3	tc4	tc5	tc6
m1	X	X				
m2	X	X			X	
m3		X				
m4			X			
m5			X			
m6			X	X		
m7				X		



Reduction of the test suite in *testooj*



Summary of results:

Minimal set of test cases (13 test cases of 216):

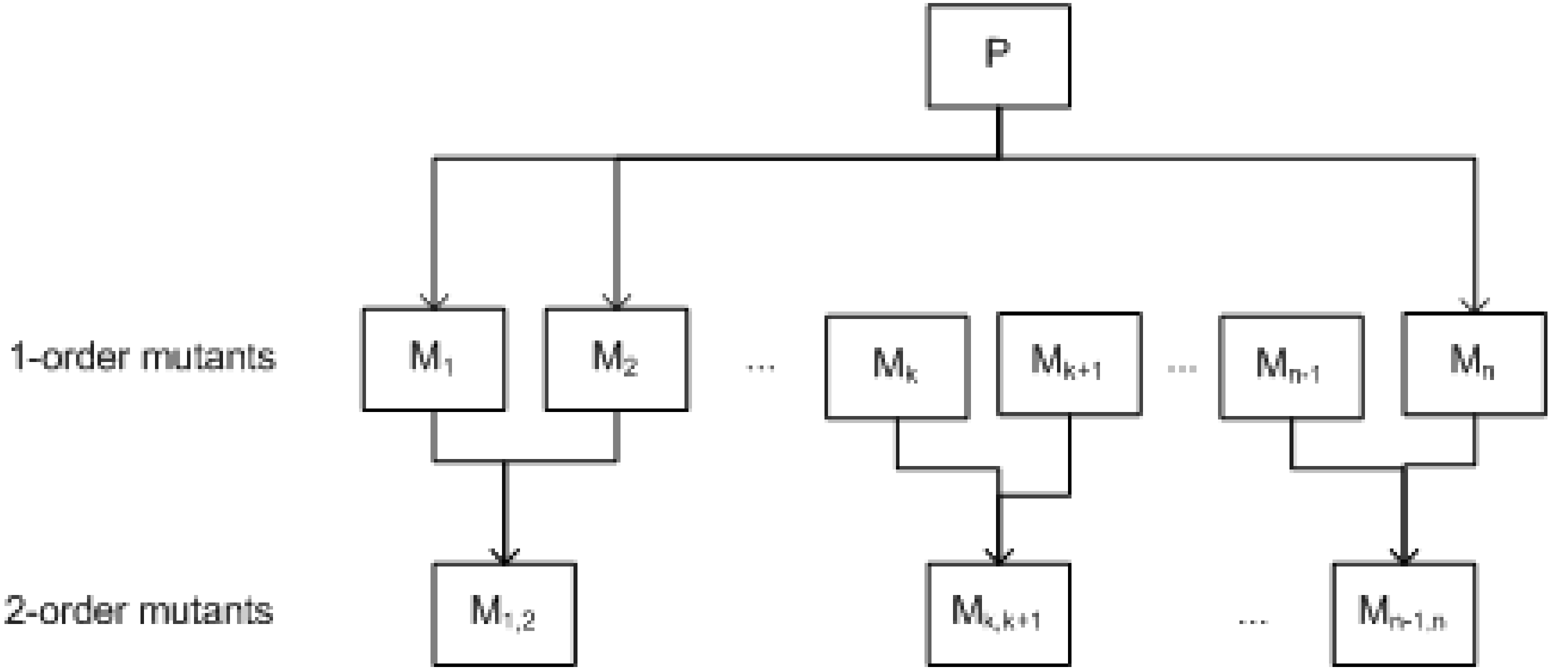
- testTS_0_95
- testTS_0_209
- testTS_0_215
- testTS_0_198
- testTS_0_99
- testTS_0_216
- testTS_0_196

Reduced suite

● .txt ○ .html

N-order mutation

- The idea is to have mutants with more than one fault

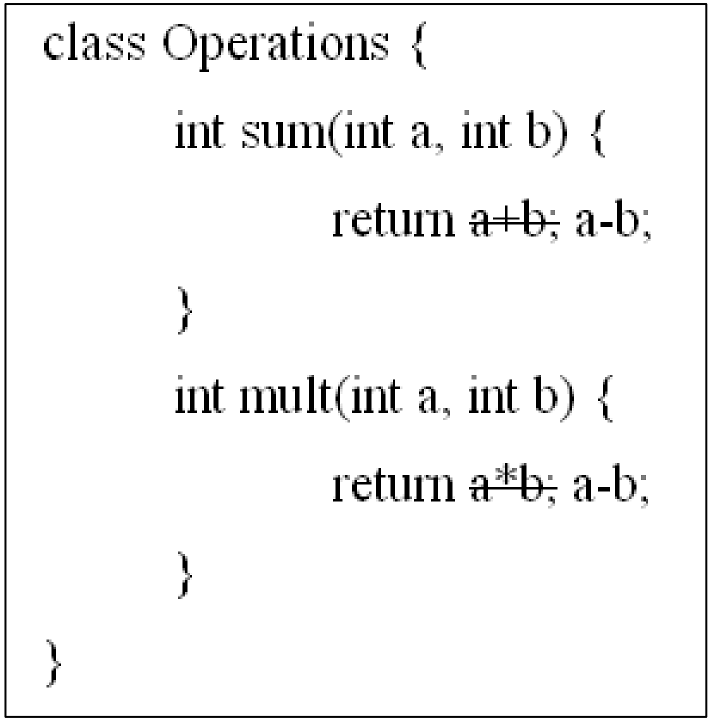


N-order mutation

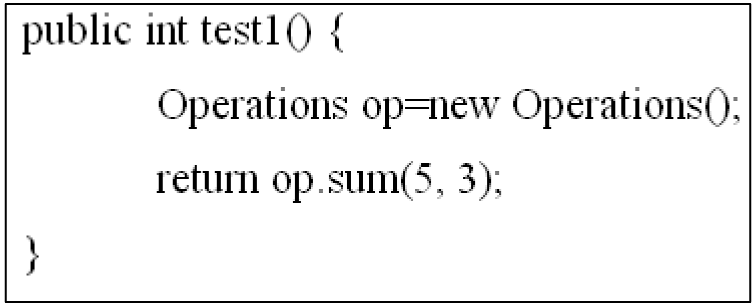
- Thus:
  - The number of mutants is closed to half the size of the original suite
  - Each equivalent mutant will be *probably* combined with a non-equivalent mutant, what implies a reduction of the number of equivalent mutants

N-order mutation

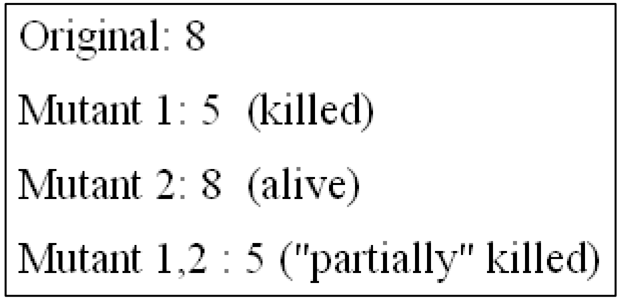
- However, there exists the possibility of having a test case that only finds one of the two faults injected



```
class Operations {
        int sum(int a, int b) {
                return ~~a+b;~~ a-b;
        }
        int mult(int a, int b) {
                return ~~a*b;~~ a-b;
        }
}
```



```
public int test1() {
        Operations op=new Operations();
        return op.sum(5, 3);
}
```



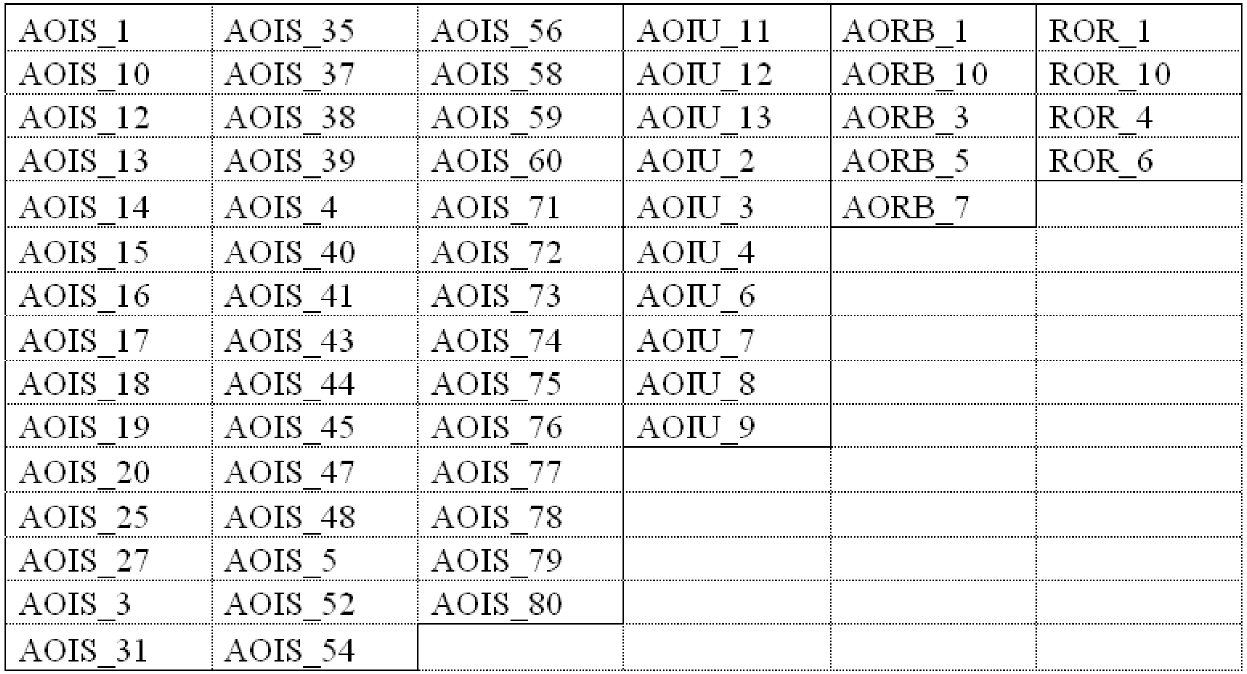
Original: 8
Mutant 1: 5 (killed)
Mutant 2: 8 (alive)
Mutant 1,2 : 5 ("partially" killed)

N-order mutation

- Algorithms for mutants combination
  - *LastToFirst*
  - *DifferentOperators*
  - *RandomMix*

LastToFirst

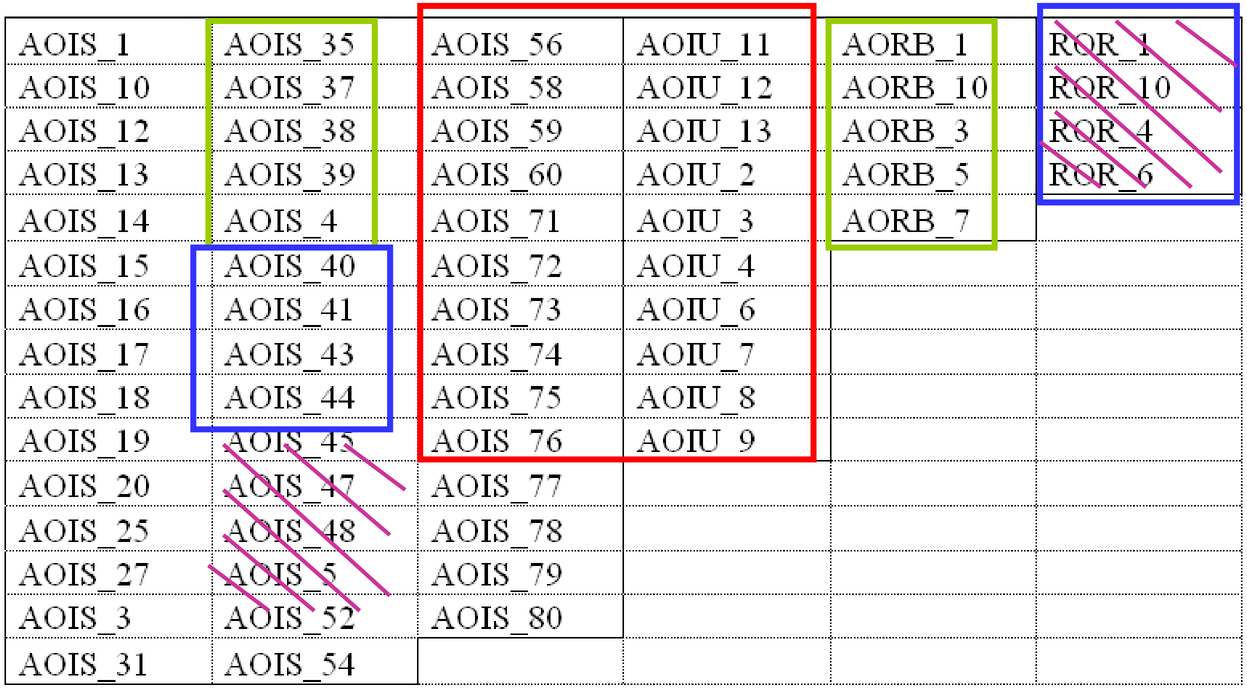
- First with the last, second with the penultime, etc.
- AOIS_1-ROR_6, AOIS_10-ROR_4, etc.
- It gets a half of the original mutant set



AOIS_1	AOIS_35	AOIS_56	AOIU_11	AORB_1	ROR_1
AOIS_10	AOIS_37	AOIS_58	AOIU_12	AORB_10	ROR_10
AOIS_12	AOIS_38	AOIS_59	AOIU_13	AORB_3	ROR_4
AOIS_13	AOIS_39	AOIS_60	AOIU_2	AORB_5	ROR_6
AOIS_14	AOIS_4	AOIS_71	AOIU_3	AORB_7	
AOIS_15	AOIS_40	AOIS_72	AOIU_4		
AOIS_16	AOIS_41	AOIS_73	AOIU_6		
AOIS_17	AOIS_43	AOIS_74	AOIU_7		
AOIS_18	AOIS_44	AOIS_75	AOIU_8		
AOIS_19	AOIS_45	AOIS_76	AOIU_9		
AOIS_20	AOIS_47	AOIS_77			
AOIS_25	AOIS_48	AOIS_78			
AOIS_27	AOIS_5	AOIS_79			
AOIS_3	AOIS_52	AOIS_80			
AOIS_31	AOIS_54				

DifferentOperators

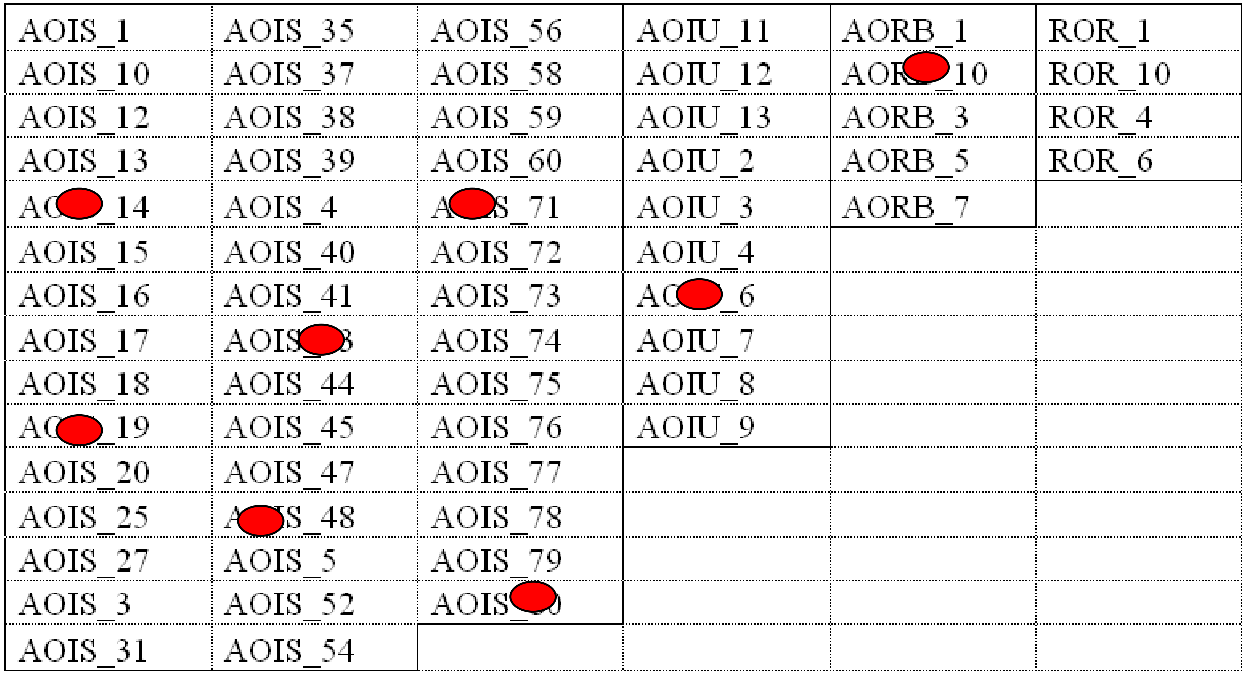
- Combines mutants obtained with different operators
- Mutants proceeding from the least frequent operator are used more than once



AOIS_1	AOIS_35	AOIS_56	AOIU_11	AORB_1	~~ROR_1~~
AOIS_10	AOIS_37	AOIS_58	AOIU_12	AORB_10	~~ROR_10~~
AOIS_12	AOIS_38	AOIS_59	AOIU_13	AORB_3	~~ROR_4~~
AOIS_13	AOIS_39	AOIS_60	AOIU_2	AORB_5	~~ROR_6~~
AOIS_14	AOIS_4	AOIS_71	AOIU_3	AORB_7	
AOIS_15	AOIS_40	AOIS_72	AOIU_4		
AOIS_16	AOIS_41	AOIS_73	AOIU_6		
AOIS_17	AOIS_43	AOIS_74	AOIU_7		
AOIS_18	AOIS_44	AOIS_75	AOIU_8		
AOIS_19	~~AOIS_45~~	AOIS_76	AOIU_9		
AOIS_20	~~AOIS_47~~	AOIS_77			
AOIS_25	~~AOIS_48~~	AOIS_78			
AOIS_27	~~AOIS_5~~	AOIS_79			
AOIS_3	~~AOIS_52~~	AOIS_80			
AOIS_31	AOIS_54				

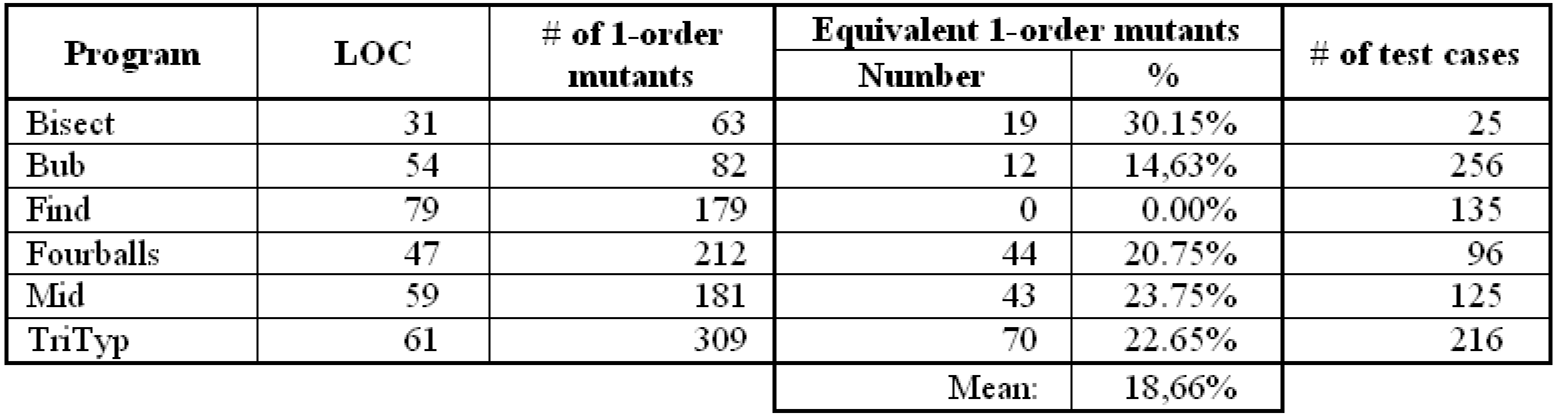
RandomMix

- Makes a pure random combination

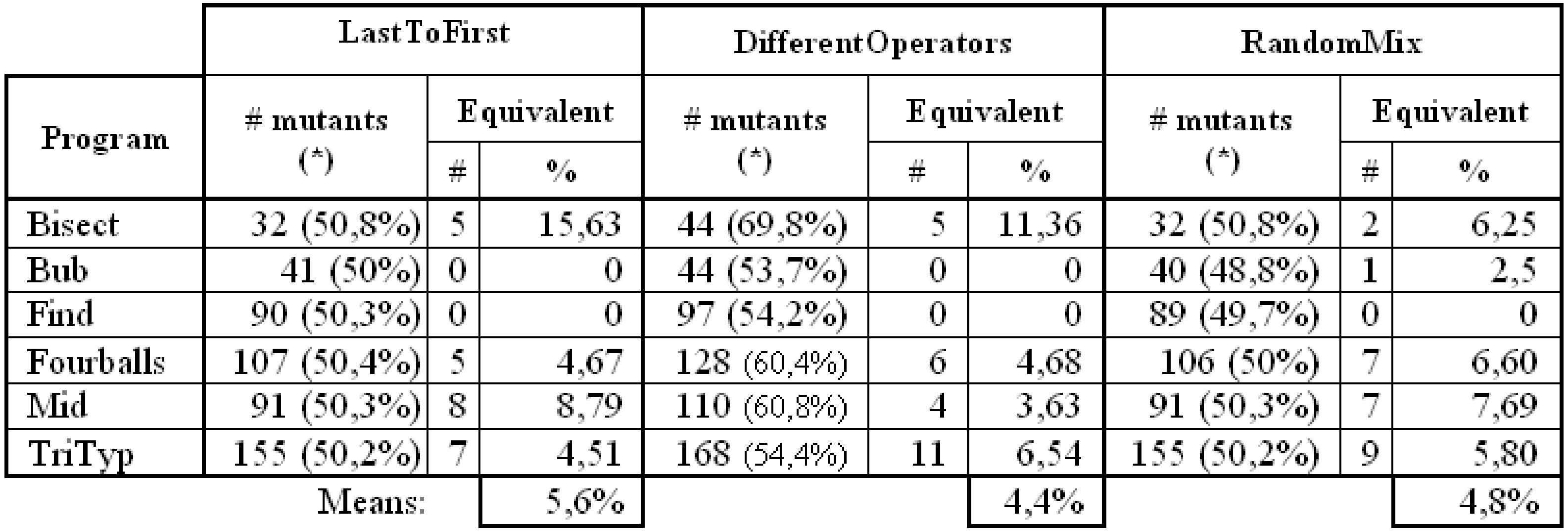


AOIS_1	AOIS_35	AOIS_56	AOIU_11	AORB_1	ROR_1
AOIS_10	AOIS_37	AOIS_58	AOIU_12	AOR●_10	ROR_10
AOIS_12	AOIS_38	AOIS_59	AOIU_13	AORB_3	ROR_4
AOIS_13	AOIS_39	AOIS_60	AOIU_2	AORB_5	ROR_6
AO●_14	AOIS_4	A●S_71	AOIU_3	AORB_7	
AOIS_15	AOIS_40	AOIS_72	AOIU_4		
AOIS_16	AOIS_41	AOIS_73	AO●_6		
AOIS_17	AOIS●3	AOIS_74	AOIU_7		
AOIS_18	AOIS_44	AOIS_75	AOIU_8		
AO●_19	AOIS_45	AOIS_76	AOIU_9		
AOIS_20	AOIS_47	AOIS_77			
AOIS_25	A●S_48	AOIS_78			
AOIS_27	AOIS_5	AOIS_79			
AOIS_3	AOIS_52	AOIS●0			
AOIS_31	AOIS_54				

Results with benchmark programs

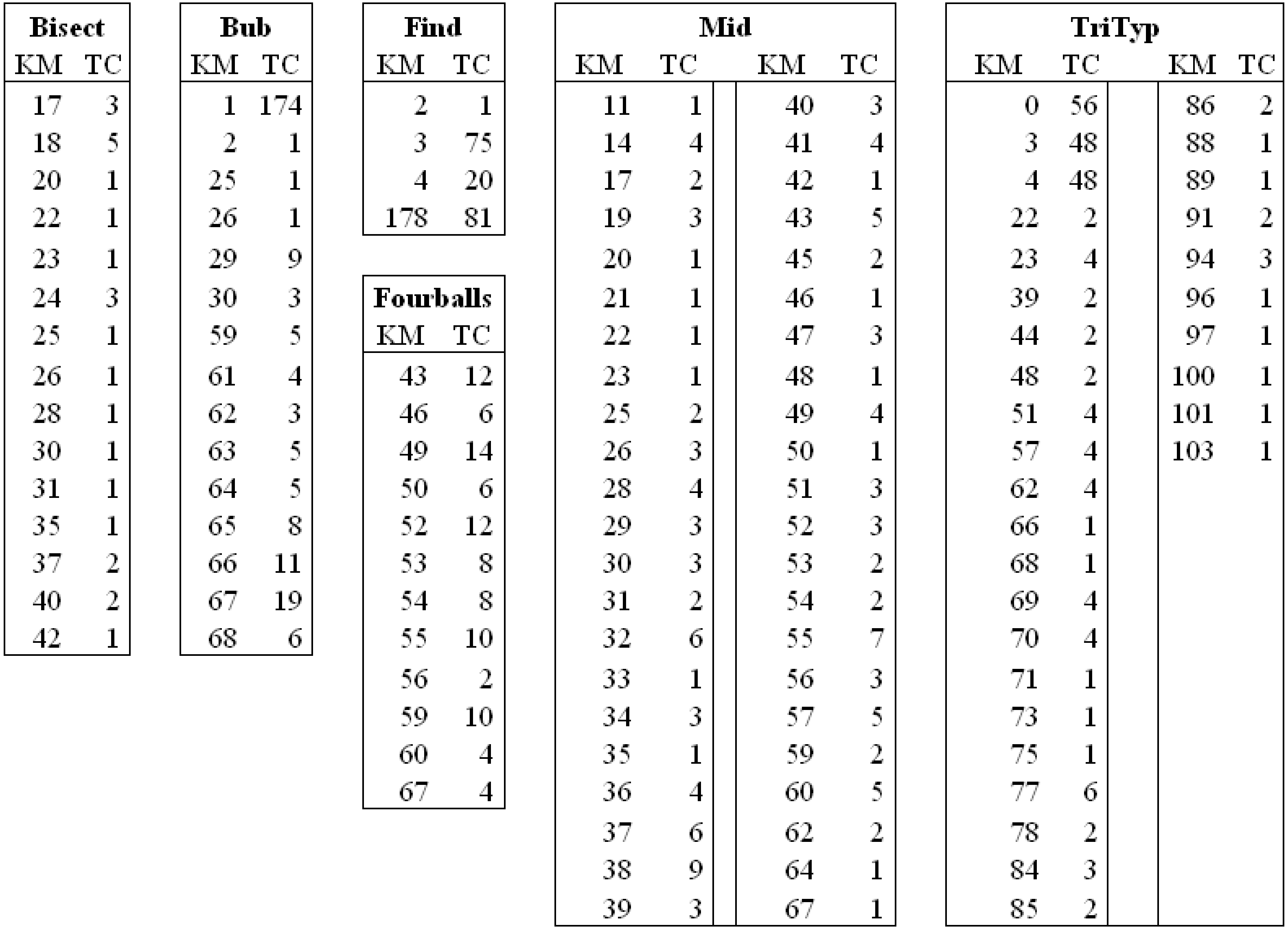


Program	LOC	# of 1-order mutants	Equivalent 1-order mutants		# of test cases
			Number	%	
Bisect	31	63	19	30.15%	25
Bub	54	82	12	14,63%	256
Find	79	179	0	0.00%	135
Fourballs	47	212	44	20.75%	96
Mid	59	181	43	23.75%	125
TriTyp	61	309	70	22.65%	216
			Mean:	18,66%	



	LastToFirst			DifferentOperators			RandomMix		
Program	# mutants (*)	Equivalent		# mutants (*)	Equivalent		# mutants (*)	Equivalent	
		#	%		#	%		#	%
Bisect	32 (50,8%)	5	15,63	44 (69,8%)	5	11,36	32 (50,8%)	2	6,25
Bub	41 (50%)	0	0	44 (53,7%)	0	0	40 (48,8%)	1	2,5
Find	90 (50,3%)	0	0	97 (54,2%)	0	0	89 (49,7%)	0	0
Fourballs	107 (50,4%)	5	4,67	128 (60,4%)	6	4,68	106 (50%)	7	6,60
Mid	91 (50,3%)	8	8,79	110 (60,8%)	4	3,63	91 (50,3%)	7	7,69
TriTyp	155 (50,2%)	7	4,51	168 (54,4%)	11	6,54	155 (50,2%)	9	5,80
	Means:		5,6%			4,4%			4,8%

Results with benchmark programs



Bisect KM	Bisect TC
17	3
18	5
20	1
22	1
23	1
24	3
25	1
26	1
28	1
30	1
31	1
35	1
37	2
40	2
42	1

Bub KM	Bub TC
1	174
2	1
25	1
26	1
29	9
30	3
59	5
61	4
62	3
63	5
64	5
65	8
66	11
67	19
68	6

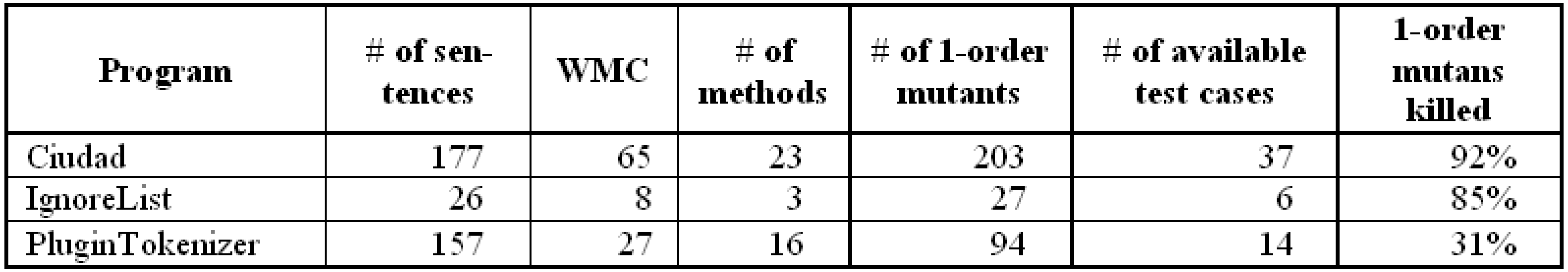
Find KM	Find TC
2	1
3	75
4	20
178	81

Fourballs KM	Fourballs TC
43	12
46	6
49	14
50	6
52	12
53	8
54	8
55	10
56	2
59	10
60	4
67	4

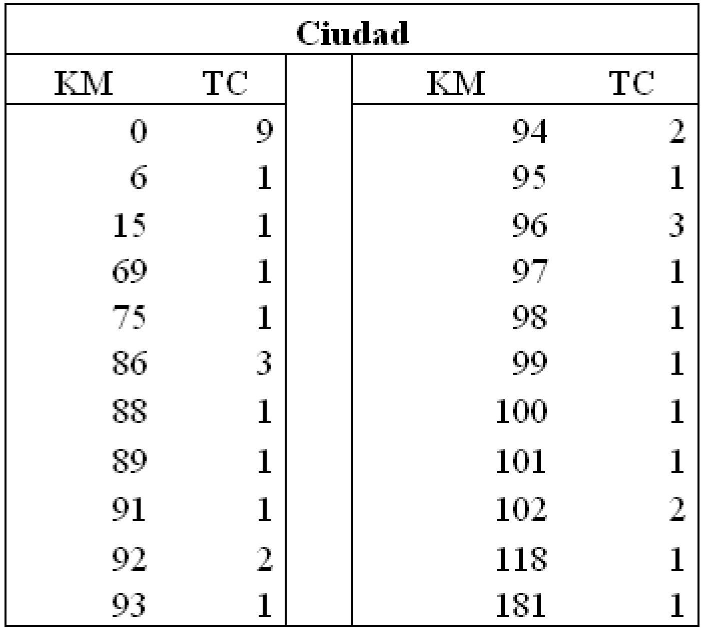
Mid KM	Mid TC	Mid KM	Mid TC
11	1	40	3
14	4	41	4
17	2	42	1
19	3	43	5
20	1	45	2
21	1	46	1
22	1	47	3
23	1	48	1
25	2	49	4
26	3	50	1
28	4	51	3
29	3	52	3
30	3	53	2
31	2	54	2
32	6	55	7
33	1	56	3
34	3	57	5
35	1	59	2
36	4	60	5
37	6	62	2
38	9	64	1
39	3	67	1

TriTyp KM	TriTyp TC	TriTyp KM	TriTyp TC
0	56	86	2
3	48	88	1
4	48	89	1
22	2	91	2
23	4	94	3
39	2	96	1
44	2	97	1
48	2	100	1
51	4	101	1
57	4	103	1
62	4		
66	1		
68	1		
69	4		
70	4		
71	1		
73	1		
75	1		
77	6		
78	2		
84	3		
85	2		

Results with industrial software

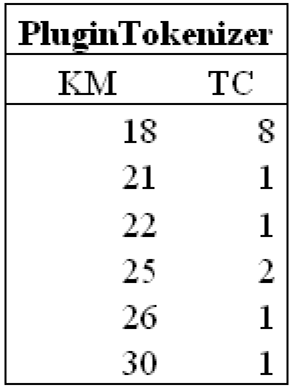


Program	# of sentences	WMC	# of methods	# of 1-order mutants	# of available test cases	1-order mutans killed
Ciudad	177	65	23	203	37	92%
IgnoreList	26	8	3	27	6	85%
PluginTokenizer	157	27	16	94	14	31%



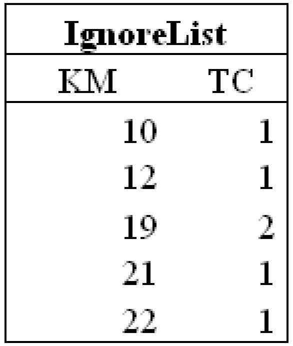
Ciudad

KM	TC	KM	TC
0	9	94	2
6	1	95	1
15	1	96	3
69	1	97	1
75	1	98	1
86	3	99	1
88	1	100	1
89	1	101	1
91	1	102	2
92	2	118	1
93	1	181	1



PluginTokenizer

KM	TC
18	8
21	1
22	1
25	2
26	1
30	1



IgnoreList

KM	TC
10	1
12	1
19	2
21	1
22	1

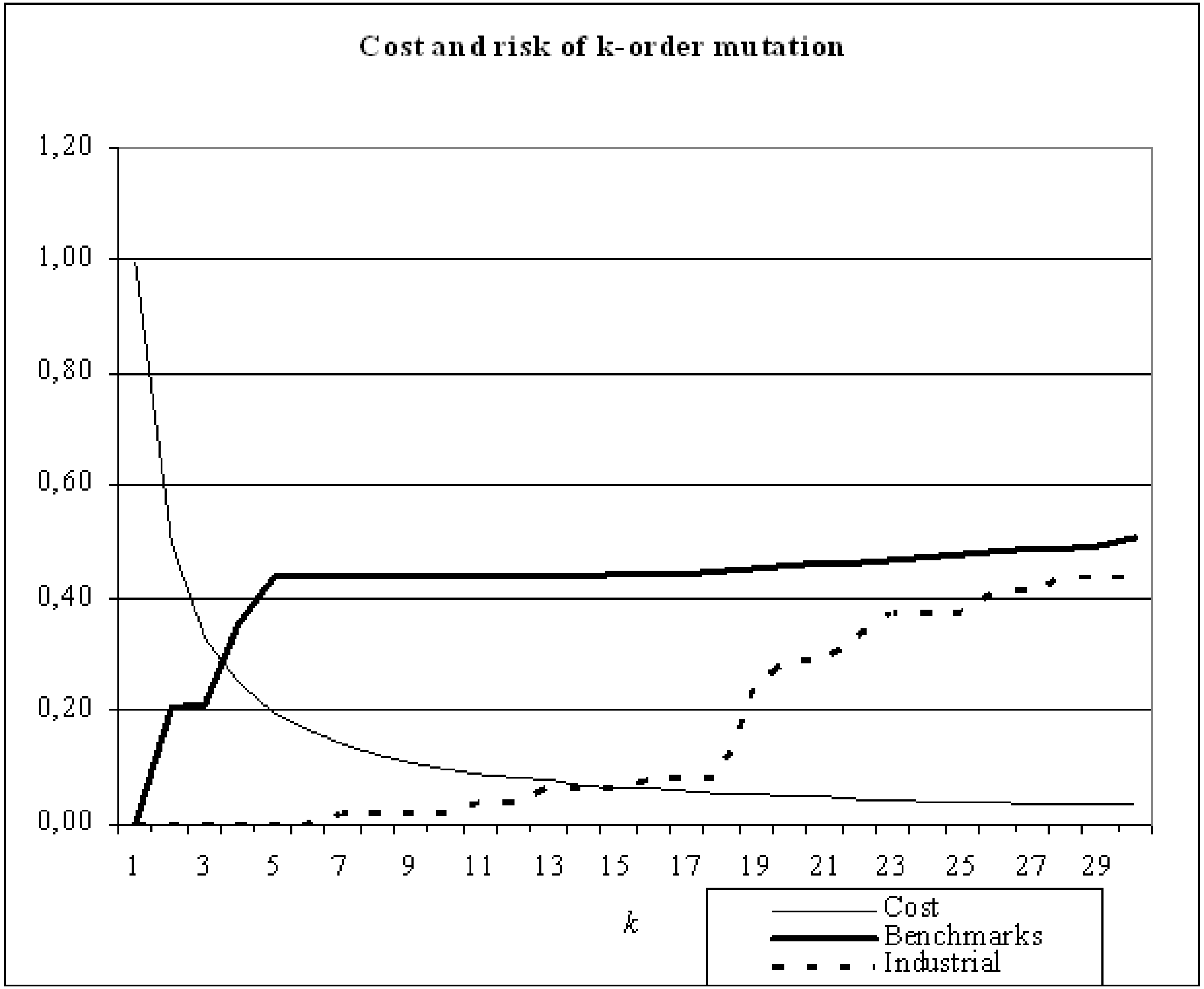
Cost-risk analysis

		Benchmark		Industrial	
K	Cost	TC	Risk	TC	Risk
1	1,00	174	0,00	0	0,00
2	0,50	2	0,21	0	0,00
3	0,33	123	0,21	0	0,00
4	0,25	68	0,36	0	0,00
5	0,20	0	0,44	0	0,00
6	0,17	0	0,44	1	0,00
7	0,14	0	0,44	0	0,02
8	0,13	0	0,44	0	0,02
9	0,11	0	0,44	0	0,02
10	0,10	0	0,44	1	0,02
11	0,09	1	0,44	0	0,04
12	0,08	0	0,44	1	0,04
13	0,08	0	0,44	0	0,06
14	0,07	4	0,44	0	0,06
15	0,07	0	0,44	1	0,06

		Benchmark		Industrial	
K	Cost	TC	Risk	TC	Risk
16	0,06	0	0,44	0	0,08
17	0,06	5	0,44	0	0,08
18	0,06	5	0,45	8	0,08
19	0,05	3	0,46	2	0,25
20	0,05	2	0,46	0	0,29
21	0,05	1	0,46	2	0,29
22	0,05	4	0,46	2	0,33
23	0,04	6	0,47	0	0,38
24	0,04	3	0,47	0	0,38
25	0,04	4	0,48	2	0,38
26	0,04	5	0,48	1	0,42
27	0,04	0	0,49	0	0,44
28	0,04	5	0,49	0	0,44
29	0,03	12	0,50	0	0,44
30	0,03	7	0,51	1	0,44

$$risk(k) = 1 - \frac{\text{\# of test cases killing } k \text{ mutants or more}}{\text{total \# of test cases}}$$

Cost-risk analysis



Conclusions

- Mutation is an excellent testing technique
- From the point of view of research, it is mature
- From the industry point of view, user-friendly tools are required
- Mutation is also applied at other levels: black-box, components, web services, models…

Mutation testing: practical aspects and cost analysis

Macario Polo and Mario Piattini
Alarcos Group
Department of Information Systems and Technologies
University of Castilla-La Mancha (Spain)

Macario.Polo@uclm.es