TÉCNICO LISBOA

Decreasing the cost of mutation testing with second-order mutants

Macario Polo, Mario Piattini and Ignacio García-Rodríguez

Group 06

CONTEXT

Mutant Testing

Designed to ensure the quality of a software testing tool

+ faults = better quality of test suite

CONTEXT

Mutant Testing

01

Mutant Generation

Generation of mutants

02

Mutant Execution

Execution of test cases

03

Result Analysis

Check mutation score

MUTATION TESTING | How does it work?

1. Mutant Generation

Program ***P*** → **mutation operator** →

Mutant ***M1***

Mutant ***M2***

Mutant ***M3***

Mutant ***M4***

...

MUTATION TESTING

How does it work?

1. Mutant Generation

Example 1

Version	Code

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++;
	}

MUTATION TESTING

How does it work?

1. Mutant Execution

		Test data (a,b)			
		(1, 1)	(0, 0)	(-1, 0)	(-1, -1)
Program versions	Original	2	0	-1	-2
	Mutant 1	0	0	-1	0
	Mutant 2	1	0	0	1
	Mutant 3	1	Error	Error	1
	Mutant 4	2	0	-1	-2

MUTATION TESTING

How does it work?

1. Result Analysis

# of killed mutants

# of equivalent (alive) mutants

MUTATION TESTING | How does it work?

1. Result Analysis

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

P : program under test
T : test suite
K : number of killed mutants
M : number of generated mutants
E : number of equivalent mutants

Figure 2. Mutation score.

Detection of # of equivalent mutants

PROBLEM

Mutation testing requires **many resources** to be successfully accomplished, and in **all of its stages**

MOTIVATION

Reduce the # of mutants?

Quality of the test suite

Decrease the cost without compromising the quality of the test suite

PREVIOUS WORK

PREVIOUS WORK

Mutant Generation

Selective Mutation

Mutant Schemata

a ARITHMETIC_OPERATOR b

PREVIOUS WORK

Mutant Execution

Vast number of test cases

Non-standard computer architectures

weak mutation

Reduction of the # of test cases

prioritize program functions

PREVIOUS WORK

Result Analysis

detection of equivalent mutants

it can only detect 50% of EM
only with constraint annotations

GOAL

Reduce the # of mutants by **combining mutant pairs** into new mutants

BASIC CONCEPTS

Coupling Effect

=

detect simple and complex faults

BASIC CONCEPTS

Example 1

Version	Code

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++;
	}

1 simple fault each

BASIC CONCEPTS

Example 1

Version	Code

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++;
	}

apply generation tool (2nd time)

BASIC CONCEPTS

1st-order mutants → 1 simple fault

2nd-order mutants → 2 simple faults

BASIC CONCEPTS

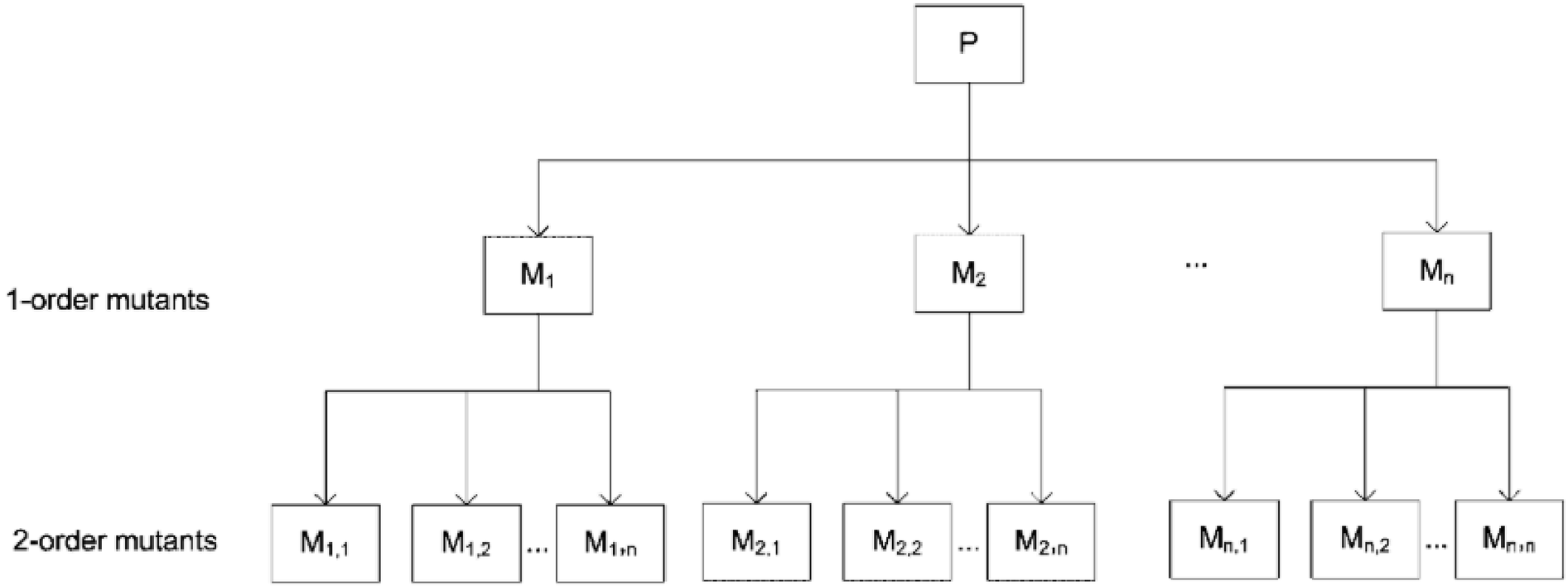


Figure 3. Exponential growth of the number of kth-order mutants.

Don’t forget!

- 3 computationally expensive steps
- All of them are dependent on one another

FORMULATION

Reduce the # of Equivalent Mutants (EM)

how? → combining pairs of 1st-order mutants

Table I. Equivalence of a second-order mutant.

M_i	M_j	$M_{i,j}$
Equivalent	Equivalent	Equivalent
Equivalent	Non-equivalent	Non-equivalent
Non-equivalent	Equivalent	Non-equivalent
Non-equivalent	Non-equivalent	Non-equivalent (very probably)

FORMULATION

Reduce the # of Equivalent Mutants (EM)

Table I. Equivalence of a second-order mutant.

M_i	M_j	$M_{i,j}$
Equivalent	Equivalent	Equivalent
Equivalent	Non-equivalent	Non-equivalent
Non-equivalent	Equivalent	Non-equivalent
Non-equivalent	Non-equivalent	Non-equivalent (very probably)

Table II. Two non-equivalent mutants may produce one equivalent mutant.

P	M_1	M_2	$M_{1,2}$

int foo(int x) {
  int $r = x + 1$;
  $r = r - 1$;
  return r;
}
	int foo(int x) {
	  int $r = x - 1$; *
	  $r = r - 1$;
	  return r;
	}
		int foo(int x) {
		  int $r = x + 1$;
		  $r = r + 1$; *
		  return r;
		}
			int foo(int x) {
			  int $r = x - 1$; *
			  $r = r + 1$; *
			  return r;
			}

FORMULATION

Reduce the # of Equivalent Mutants (EM)

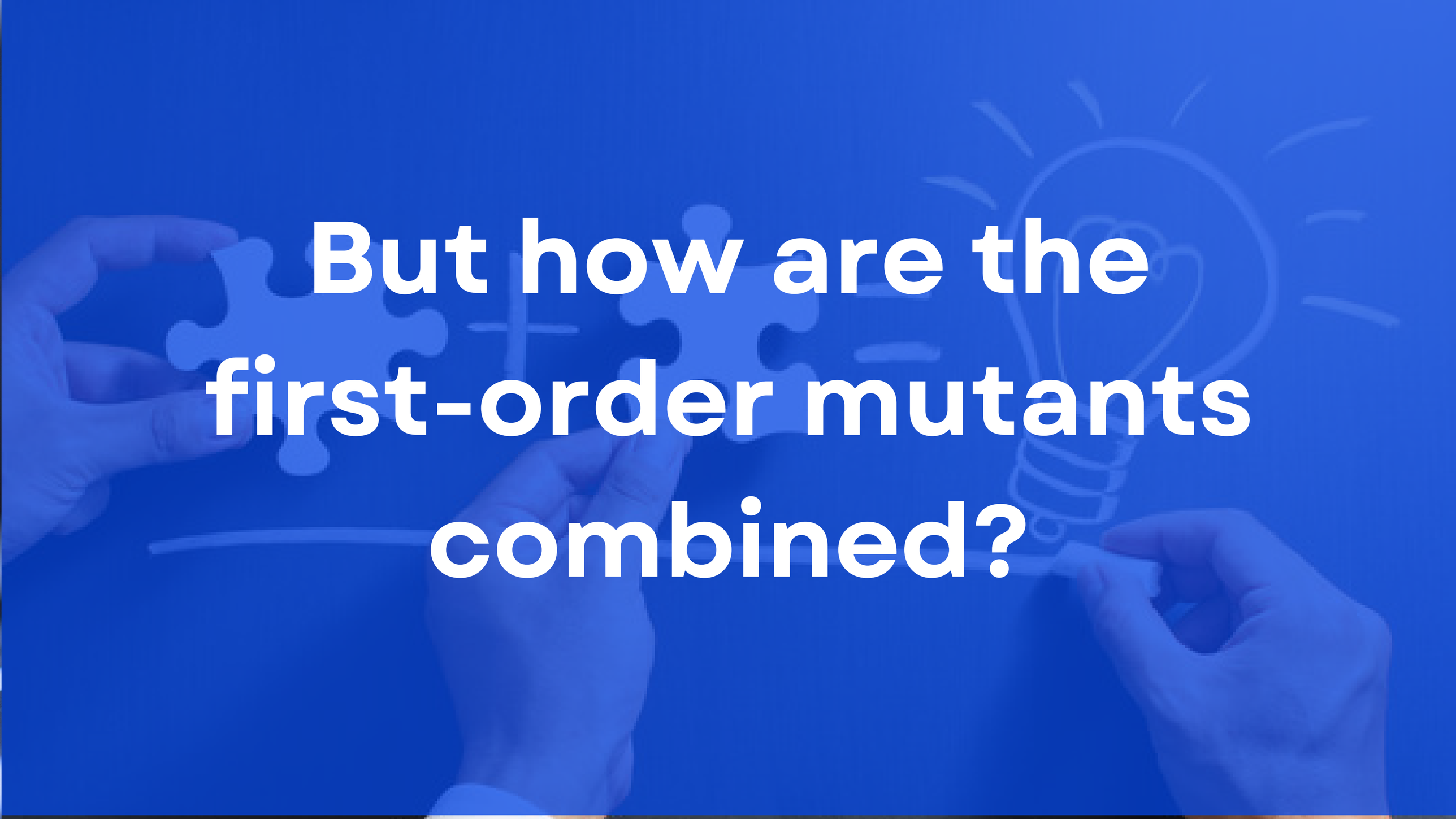
$$T \text{ is mutation - adequate for } M \Rightarrow T \text{ is mutation - adequate for } M'$$

Figure 5. Mutation adequacy.

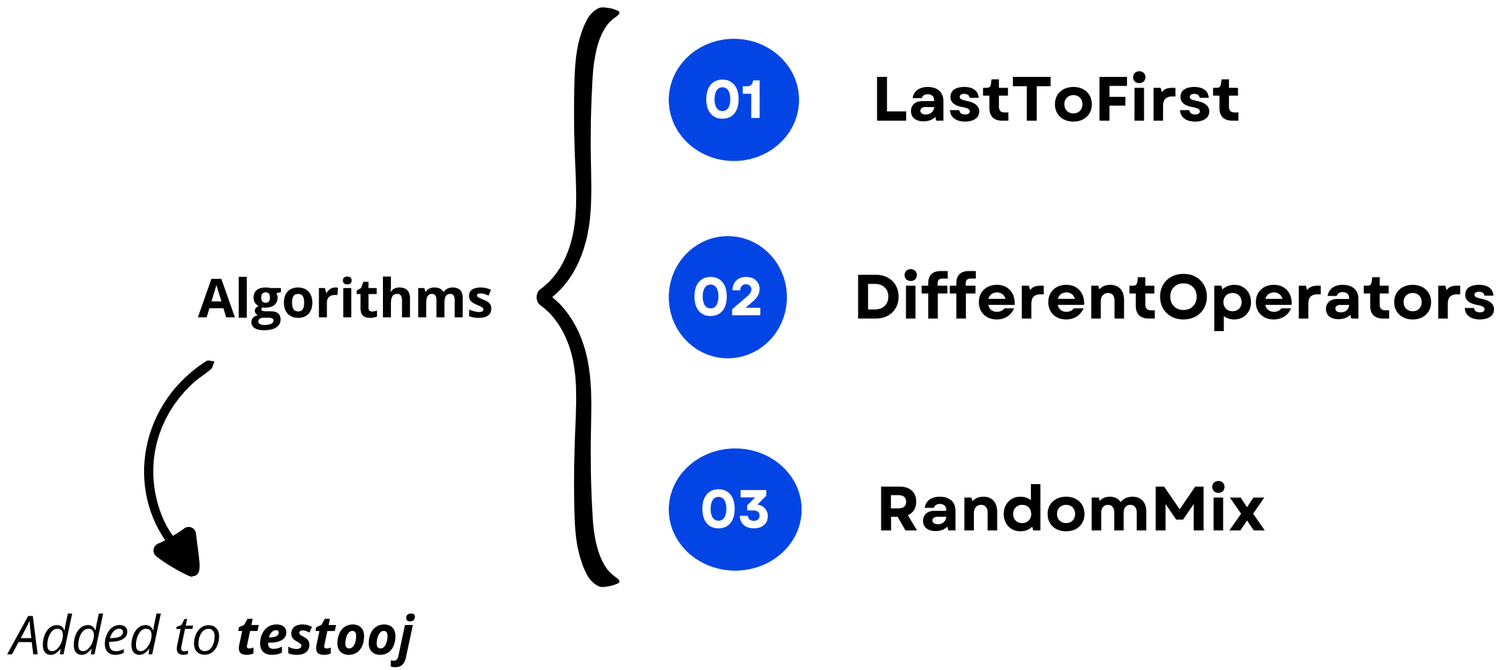
Don’t forget

Mutation Testing

- Ensures the quality of a software testing tool
- 3 computationally expensive steps
- Reduces the cost by mutant combination


But how are the
first-order mutants
combined?

ALGORITHMS FOR 2ND-ORDER MUTANT GENERATION



ALGORITHMS FOR 2ND-ORDER MUTANT GENERATION

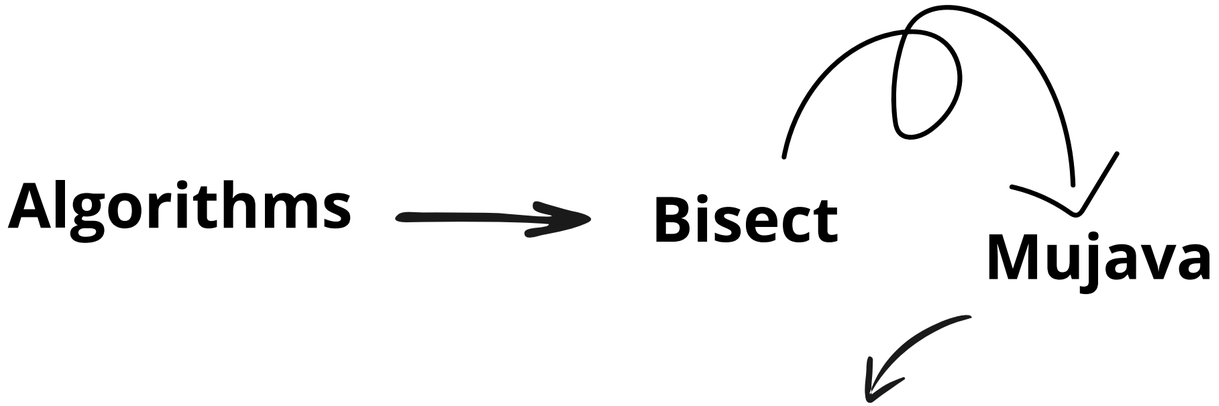


Table III. MuJava mutants generated for the *Bisect* program.

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				

63 mutants !

ALGORITMHS

LastToFirst

Table III. MuJava mutants generated for the *Bisect* program.

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				

AOIS_1ROR_6

AOIS_10ROR_4

reduces mutants by half

Simplicity

ALGORITMHS

Different Operators

Table III. MuJava mutants generated for the *Bisect* program.

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				

AOIS produces more EM

AOIS ✗ AOIS

AOIS ✓ AOUI
AORB
ROR

Minimum # of 2nd order mutants

=

Operator produces more mutants

ALGORITMHS

RandomMix

Serve as a baseline

Table III. MuJava mutants generated for the *Bisect* program.

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				

Each mutant is used once

except

Odd number of mutants

EVALUATION

Experiment 1: benchmark programs

Decrease the cost without compromising the quality of the test suite

EVALUATION

Experiment 1: benchmark programs

01 **Bisect**

02 **Bub**

03 **Find**

04 **Fourballs**

05 **Mid**

06 **TriTyp**

EVALUATION | Experiment 1: benchmark programs

Experimental Setup

all test suites were ***mutation-adequate***

Mutant Generation

MuJava

Mutant Execution

testooj

2nd-order Mutants Generation

3 algorithms

EVALUATION | Experiment 1: benchmark programs

Results - Motivation Part 1 Reduce the cost

Table V. First-order mutants generated for the benchmark programs.

Program	LOC	No. of first-order mutants	Equivalent first-order mutants		No. 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	

EVALUATION | Experiment 1: benchmark programs

Results - Motivation Part 1 Reduce the cost

Table VI. Second-order mutants after combining with the three algorithms.

	LastToFirst			DifferentOperators			RandomMix		
		Equivalent			Equivalent			Equivalent	
Program	No. of mutants*	No.	%	No. of mutants*	No.	%	No. of mutants*	No.	%
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
	Mean		5.6			4.4			4.8

*Values in parentheses represent the percentage of second-order mutants with respect to first-order mutants.

EVALUATION

Experiment 1: benchmark programs

Results - Motivation Part 2

Guaranty quality of tests

Table VIII. Number of first-order mutants killed (KM) by test cases (TC) in the benchmark programs.

Bisect		Bub		Find		Mid				TriTyp			
KM	TC	KM	TC	KM	TC	KM	TC	KM	TC	KM	TC	KM	TC
17	3	1	174	2	1	11	1	40	3	0	56	86	2
18	5	2	1	3	75	14	4	41	4	3	48	88	1
20	1	25	1	4	20	17	2	42	1	4	48	89	1
22	1	26	1	178	81	19	3	43	5	22	2	91	2
23	1	29	9			20	1	45	2	23	4	94	3
24	3	30	3	Fourballs		21	1	46	1	39	2	96	1
25	1	59	5	KM	TC	22	1	47	3	44	2	97	1
26	1	61	4	43	12	23	1	48	1	48	2	100	1
28	1	62	3	46	6	25	2	49	4	51	4	101	1
30	1	63	5	49	14	26	3	50	1	57	4	103	1
31	1	64	5	50	6	28	4	51	3	62	4		
35	1	65	8	52	12	29	3	52	3	66	1		
37	2	66	11	53	8	30	3	53	2	68	1		
40	2	67	19	54	8	31	2	54	2	69	4		
42	1	68	6	55	10	32	6	55	7	70	4		
				56	2	33	1	56	3	71	1		
				59	10	34	3	57	5	73	1		
				60	4	35	1	59	2	75	1		
				67	4	36	4	60	5	77	6		
						37	6	62	2	78	2		
						38	9	64	1	84	3		
						39	3	67	1	85	2		

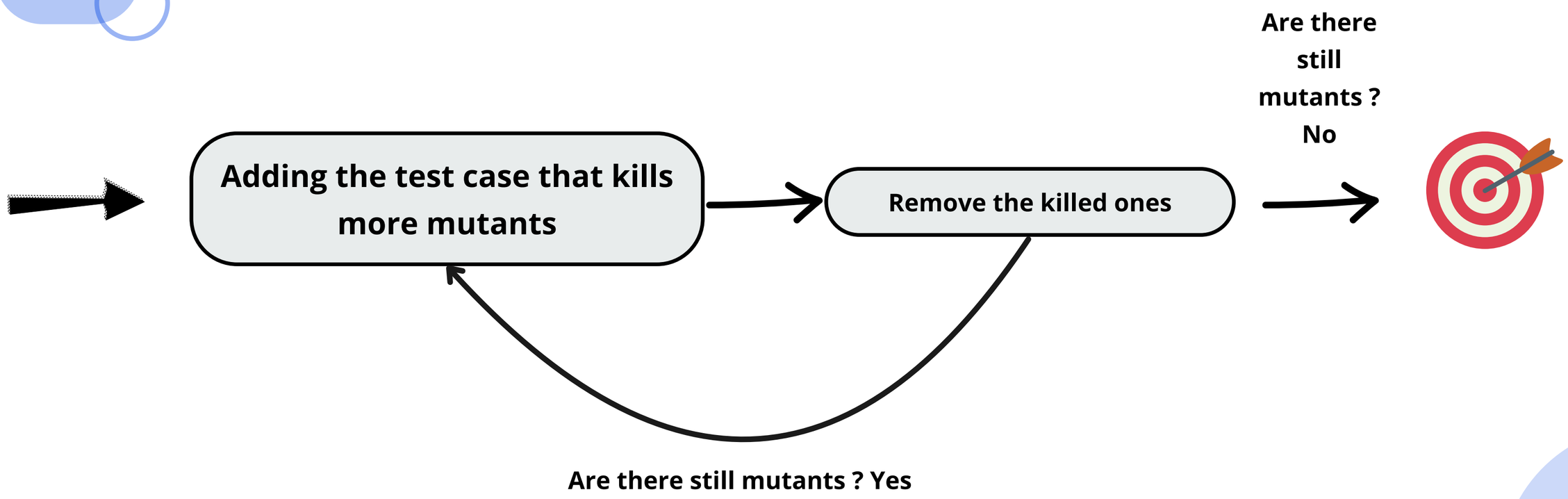
EVALUATION | Experiment 1: benchmark programs

Analysis with test suite reduction algorithm

Main focus —understand→ **how well the reduced test suites perform ?**

EVALUATION | Experiment 1: benchmark programs

Reduction Algorithm



EVALUATION

Experiment 2: industrial software

3 classes from 3 industrial systems

evaluate →

quality of the existing test cases

EVALUATION | Experiment 2: industrial software

The 3 selected classes

System

Compatible with Mujava and testooj

01 **Ciudad**

02 **Ignore List**

03 **PluginTokenizer**

test cases exercise functionallities

EVALUATION | Experiment 2: industrial software

Results

Table XI. Some quantitative data of the industrial programs.

Program	No. of sentences	WMC	No. of methods	No. of first-order mutants	No. of available test cases	First-order mutants killed (%)
Ciudad	177	65	23	203	37	92
IgnoreList	26	8	3	27	6	85
PluginTokenizer	157	27	16	94	14	31

High mutation score → Effective test cases

Low mutation score → need to be improved test cases

COST/ RISK ANALYSIS OF *K*TH-ORDER MUTATION

Could higher-order mutations further reduce costs?

COST/ RISK ANALYSIS OF *K*TH-ORDER MUTATION

Data Analysis

Table XIV. Relative cost and risk of *k*th-order mutation.

		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
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

COST/ RISK ANALYSIS OF *K*TH-ORDER MUTATION

Data Analysis

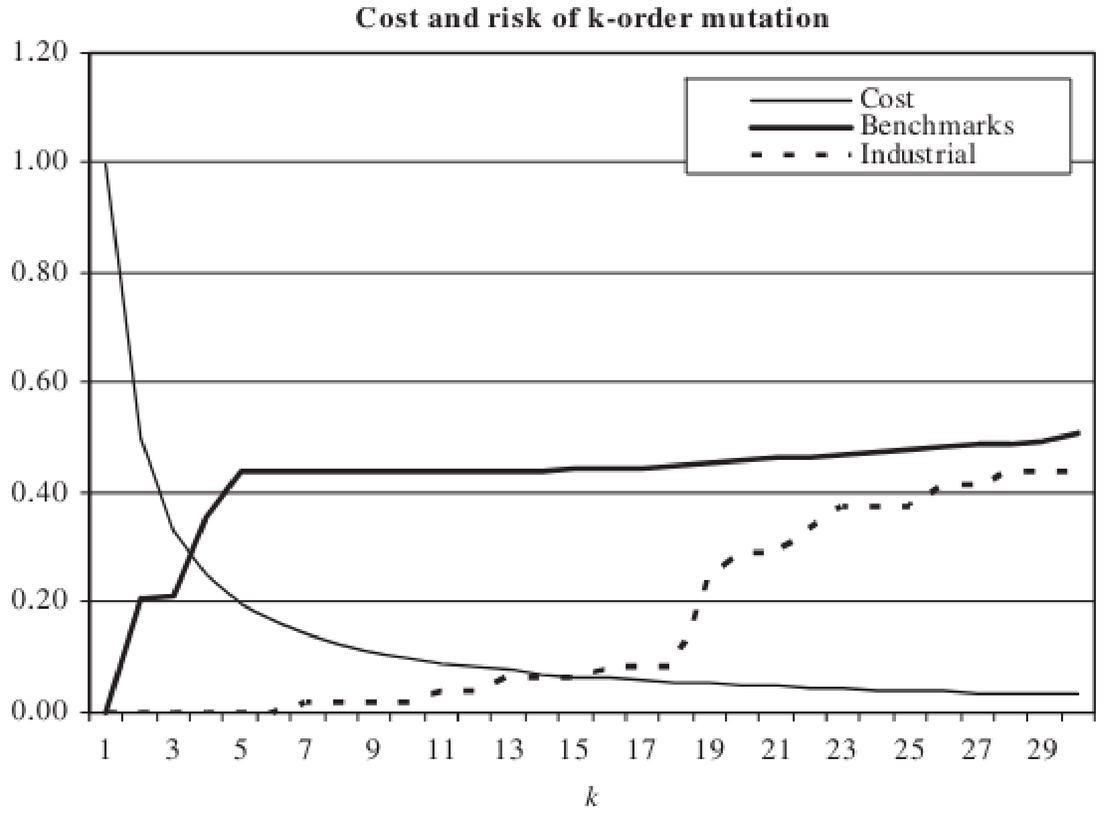


Figure 9. Relationship between cost and risk of *k*th-order mutation.

THREATS TO VALIDITY

Threats

- **Threats to Construct Validity:** How accurately variables are measured
- **Threats to Internal Validity:** Cofidence in establishing a cause-effect relationship
- **Threats to External Validity:** Examines the generalizability of research results

CONCLUSION

What are the contributions?

- **Reduce the cost** of mutation testing

CONCLUSION

What are the achievements?

- Reduces the # of mutants by **50%**

- Reduces the # of equivalent mutants to **5%**

3 combination algorithms

DifferentOperators **reduces mutants to 40%**

CONCLUSION

What are future directions?

- Further validation with **Industrial Software**
- Explore **higher-order mutants**

CONCLUSION

Our analysis

- ✅ Good idea that benefits cost reduction
- ❌ Complex higher-order mutants in real-world applicability

TÉCNICO LISBOA

THANK YOU

Mutation Testing

- Ensures the quality of a software testing tool
- 3 computationally expensive steps
- Reduces the cost by mutant combination

Group 06