Is Mutation Testing Ready to Be Adopted Industry-Wide?

Jakub Možucha and Bruno Rossi(✉)

Faculty of Informatics, Masaryk University, Brno, Czech Republic
{jmozucha,brossi}@mail.muni.cz

Abstract. Mutation Testing has a long research history as a way to improve the quality of software tests. However, it has not yet reached wide consensus for industry-wide adoption, mainly due to missing clear benefits and computational complexity for the application to large systems. In this paper, we investigate the current state of mutation testing support for Java Virtual Machine (JVM) environments. By running an experimental evaluation, we found out that while default configurations are unbearable for larger projects, using strategies such as selective operators, second order mutation and multi-threading can increase the applicability of the approach. However, there is a trade-off in terms of quality of the achieved results of the mutation analysis process that needs to be taken into account.

Keywords: Software mutation testing · Experimentation · Equivalent mutants · Selective mutation operators · Cost-reduction strategies

1 Introduction

Large amount of resources are wasted yearly due to bugs introduced in software systems, making the testing process one of the critical phases of software development [2]. A recent research reported the cost of software debugging up to a yearly \$312 Billion, with developers utilizing 50 % of their allocated time to find and fix software bugs [1]. Software Engineering is for long time striving to find ways to reduce such inefficiencies, with the constant challenge to build more robust software. Mutation Testing is one such ways, representing a powerful technique to evaluate and improve the quality of software tests written by developers [7,14].

The main idea behind Mutation testing is to create many modified copies of the original program called *mutants* — each mutant with a single variation from the original program. All mutants are then tested by test suites to get the percentage of mutants failing the tests. It has been proven that mutation testing can bring several benefits to complement the applied testing practices, e.g. for test cases prioritization [6].

However, mutation testing has been often reported to struggle to be introduced in to real-world industrial contexts [8,11,15]. So why is mutation testing not widely adopted within industry? According to Madeyski et al. [9], mainly due to (a) performance reasons, (b) the equivalent mutants problem — syntactically but not semantically equal mutants — and (c) missing integration tools. In our view, the biggest drawback of mutation testing — its great computational costs — prevented until recently to include mutation testing into the development cycle of most companies. This resulted in development of many techniques to reduce the costs of mutation testing. Furthermore, another perceived drawback might be that the advantages of running mutation testing might not be fully clear as opposed to other simpler testing approaches.

Problem. The applicability of Mutation Testing to real-world project is far from reaching consensus [4,8,9,11,15]. While it seems that improvements have been done in tools integration, performance and equivalent mutants concerns still remain the most relevant issues, and call for further analyses.

Contribution. We report on an experiment addressed at understanding the current performance of Mutation Testing in Java Virtual Machine (JVM) environments, based on our previous experience on empirical studies [16] and the needs for more industry-academia cooperation [3]. With the collaboration of an industrial partner, we are in particular looking at different strategies that can reduce runtime overhead of Mutation Testing. Among the results, we provide indications about selective operators efficiency for Mutation Testing and their impact on performance. Practitioners can gain more insights about the performance / quality trade-offs in running mutation testing by evaluating several cost reduction strategies on a typical set of projects. Such information can be relevant for the integration in their own software development process. Furthermore, we make available the experimental package for replications.

The paper is structured as follows. Section 2 reports about the background on mutation testing. In Sect. 3, we refer about the experimental evaluation, describing the experimental design, choices made, results, and threats to validity. Section 4 provides related works that evaluated mutation testing in an experimental setting. Section 5 provides the discussions and Sect. 6 the conclusions.

2 Mutation Testing Background

Mutation Testing has undergone several decades of extensive research. First formed and published by DeMillo et al. in a 1971's seminal paper [5], Mutation Testing was introduced as a technique that can help programmers to improve the tests quality significantly. The core of Mutation Analysis is creating and killing mutants. Each mutant is a copy of original source code modified (*mutated*) with a single change (*mutation*). These mutations are done based on set of predefined syntactic rules called *mutation operators*. Traditional mutation operators consist of statement deletions (e.g. removing a `break` in a loop), statement modifications (e.g. replacing a `while` with `do-while`), Boolean expression modification (e.g. switching a `!=` logical operator to `==`), or variables/constants replacements. These mutation operators can be considered to be traditional mutation operators and are mostly language independent. There are also language-dependent mutation operators that are used to mutate language-specific constructs, taking into account aspects such as encapsulation, inheritance and polymorphism.

Tests are then executed on the mutants and the failure of mutants is expected. When the tests fail, the mutant is considered killed and no further tests are needed to be run using this mutant. For example, the original Java code in Algorithm 1 is mutated using a mutation operator, which replaces == with != and produces the mutant in Algorithm 2.

Algorithm 1. Original Code

```
if (a == b) then
    // do something
else
    // do something
```

Algorithm 2. Mutated Code

```
if (a != b) then
    // do something
else
    // do something
```

If any mutant does not cause the tests to fail, it is considered live. This can have two meanings: that the tests are not sensitive enough to catch the modified code or that this mutant is equivalent. An *equivalent mutant* is syntactically different from the original, but its semantics are the same and therefore it is impossible for tests to detect them. The final indication of tests quality is *mutation score*, that is the percentage of non-equivalent mutants killed by the test data or in other terms, the number of killed mutants over the number of *non-equivalent* mutants generated.

A test data is considered mutation-adequate [12] if its mutation score is 100 %. The higher mutation score is, the more faults were discovered — therefore the better the test process. This process leads to iterative improvement of testing, moreover, inspecting live mutants can lead to discovery and resolution of other source code issues. The most serious problem with equivalent mutants is the distortion of the mutation score: software tools include them in the computation, as their accurate detection is undecidable and can only be performed by manual inspection [9].

Mutation testing is a powerful technique, but has great computational costs. In fact, these costs have prevented mutation testing to be used in a practical way for many years, despite the relatively long history of mutation testing related research. In general, these are the most expensive phases:

1. **Mutant Generation** – aside from great computational costs, also the memory consumption is considerably high in this phase. Mutation operators have to be applied on the original code and mutants have to be stored;
2. **Mutant Compilation** – phase in which the generated mutants have to be compiled. This phase can be very costly for larger programs;
3. **Execution of tests** – for every mutant, the tests have to be executed until they are not killed. Most costly are live mutants, because every test has to be run on them;

There are several approaches that have been proposed to reduce the equivalent mutation problem ([9] provides an extensive review in the area).

At the same time, the problem is very often linked to performance optimization, as less equivalent mutants generated lead to a reduction in the three phases of mutant generation, mutant compilation, and tests execution. For this reason, various cost reducing strategies were developed.

In this paper we look into several of these strategies and their applicability to improve the performance for industrial applicability. A first strategy is the *Selective Mutation* technique — the idea is to use only the mutation operators that produce statistically less equivalent mutants than others. This approach allows to reduce not only equivalent mutants, but also to improve the performance. The aim of Selective Mutation is to achieve maximum coverage by generating the least number of mutants. Complexity reduction of mutants generation is from quadratic ($O(Refs * Vars)$) to linear ($O(Refs)$) [13], while retaining much of effectiveness of non-selective mutation.

Another strategy we adopt in the current paper is *Higher Order Mutation (HOM)*. Taking into account the original mutants we discussed so far as First Order Mutants (FOMs), the technique creates mutants with more than a single mutation, referred as higher order mutants as combination of several FOMs [6]. We look in particular at four different algorithms (*Last2First*, *DifferentOperators*, *ClosePair* and *RandomMix*) implemented in Judy [9,10] to combine FOMs into Second Order Mutants (SOMs) — in which two mutants are combined:

- *Last2First* – the first mutant in the list of FOMs is combined with the last mutant in the list, the second mutant with next to last and so on;
- *DifferentOperators* – only FOMs generated by different mutation operators are combined;
- *ClosePair* – two neighboring FOMs from the list are combined;
- *RandomMix* – any two mutants from the FOMs are combined;

All these strategies use a list of first order mutants (FOM) and should generate at least 50 % less mutants, with impact on the final mutation score [9].

3 Experimental Evaluation

We designed an exploratory experiment aimed at getting insights about the current applicability of mutation testing in industrial context (summary of the overall process, Fig. 1). We run in parallel a literature review (1) and an exploratory analysis about the usage of the tools for Mutation Testing (5). The selection of the tools for the experimentation (2), as well as the experimental units (3) were done based on the criteria of the company. We designed the research questions (6) and created the experimental design (7) based on the results from the exploratory analysis, taking both into account the company's needs and theoretical constraints and aspects worth investigation from theory. Experiments were then run (8) and results provided to the industrial partners for knowledge transfer and identification of future works. Based on an exploratory pre-experiment phase, we set the following research questions:

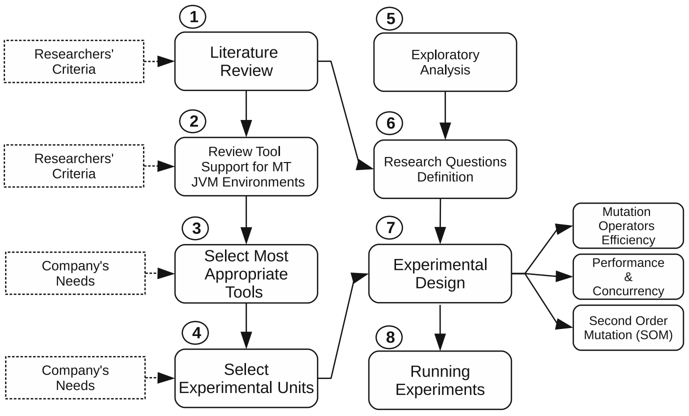

1
Researchers' Criteria
Literature Review
2
Researchers' Criteria
Review Tool Support for MT JVM Environments
3
Company's Needs
Select Most Appropriate Tools
4
Company's Needs
Select Experimental Units
5
Exploratory Analysis
6
Research Questions Definition
7
Experimental Design
Mutation Operators Efficiency
Performance & Concurrency
Second Order Mutation (SOM)
8
Running Experiments

Fig. 1. Research process workflow

- **RQ1.** What is the performance of Mutation Testing by taking into account standard configurations (i.e. no selective operators and no mutation strategy)?
- **RQ2.** What is the impact of Selective Operators on Mutation Testing efficiency & performance?
- **RQ3.** What is the impact of Second Order Mutation strategies on Mutation Testing efficiency & performance?

Given the selection of the tools for mutation testing supported by the industrial partner, we looked specifically at three areas of experiments according to the three research questions set:

EXP1. Mutation Operators Efficiency Experiments. Looking at the selection of the most efficient mutation operators that then be evaluated in the performance experiments;

EXP2. Performance Experiments & Concurrency Experiments. Looking at the single-thread and multi-threading performance of the tools with standard configurations and according to different selective operator strategies;

EXP3. SOM Experiments. Evaluating the impact of different Second Order mutation strategies (*Last2First*, *DifferentOperators*, *ClosePair*, *RandomMix*).

We run an initial review of the tools available for mutation testing in JVM environments, omitting experimental tools. We overall considered seven tools: MuJava, PITest, Javalanche, Judy, Jumble, Jester, MAJOR, that we compared according to several characteristics (Table 1). To speed-up the mutation generation process, it is now a standard the support of byte-code mutation — mutants are applied at the byte code level. The industrial partner involved in the experimentation considered PITest and Judy more relevant for a series of reasons, in particular the availability of plugins and general easiness of integration, as well as the open source license and support for Java 8. Both software were used to run the experiments.

Table 1. JVM mutation testing tools

	MuJava	PITest	Javalanche	Judy	Jumble	Jester	MAJOR
State	active	active	–	active	active	–	active
License	Apache 2.0	Apache 2.0	LGPL	BSD	GPL	open	?
Java vers.	7	5 – 8	5, 6	6 – 8	6 –8	6	7
Unit test. framework	jUnit4	jUnit4, TestNG6	jUnit4	jUnit4	jUnit4	jUnit3	jUnit4
Production tools	GUI, Eclipse	3^{rd} part. plugins	Eclipse plugin	Cmd	Cmd	Cmd	Ant
Automated class/test selection	Only classes	Yes	Yes	Yes	Only tests	Yes	Yes
Mutation Operators	method class	method	method concurr	meth. class	meth	meth	method self-def.
Byte-code mutation	Yes	Yes	Yes	Yes	Yes	–	Yes

For experimentation, we used as experimental projects libraries suggested by the industrial partner (Table 2): various Apache Commons projects and the JodaTime library. The selection was done so that the chosen projects contain possibly the most different distribution of size types, tests duration and test coverage. However, we couldn't include larger projects due to the high complexity of running mutation tests. From an initial list, we had to discard other projects that had either no tests (Apache Commons Daemon) or in which test cases were failing with either PITest or Judy (Apache Commons Compression, BeanUtils).

We included in Table 2 Mutation Size, a metric that can give indication of the complexity of the mutation process. While previous research has proved that the number of mutants is proportional to the number of variable references times the number of data objects *(O(Refs * Vars))* [12]. However, it is uneasy to determine the number of variable references for such large projects. Taking into account that modern mutation testing tools are creating mutants and running tests based on code coverage, Mutation Size is computed as coverage times size of project, as a measure of the complexity of the mutation testing process $MS = coverage * KLOC$.

3.1 Experimental Procedure

The first set of experiments was done with both tools using their default settings. The only modification was the configuration to the same number of threads, as Judy runs by default in parallel, while PITest uses only one thread.

After experiments with default settings, the experiments on mutation operators efficiency were done for each tool. The aim of these tests was to reduce the number of active operators selecting only the most efficient operators — the ones that produce mutants that are not so easy-to-kill.

After the selection of mutation operators, another set of performance tests was done using only the the selected operators. Concurrency tests were done on various number of used threads, comparing time and memory usage (PITest).

Table 2. Projects considered for the experimental evaluation. NCL = Number of Classes, KLOC = Lines of Code (thousands), NTCL = Number of Test Classes, TKLOC = Test Lines of Code (thousands), TT = Test Time, CC = Code Coverage, MS = Mutation Size

Project	Ver.	NCL	KLOC	NTCL	TK-LOC	TT	CC	MS
Apache Commons Chain	1.2	55	9.852	37	7.398	1.17	66.68 %	6.57
Apache Commons CLI	1.3.1	23	6.161	25	5.214	1.39	96.38 %	5.94
Apache Commons Codec	1.10	60	15.869	55	15.042	5.31	95.01 %	15.08
Apache Commons CSV	1.2	11	3.515	15	3.821	1.76	94.00 %	3.30
Apache Commnos DbUtils	1.6	30	7.611	26	4.453	1.36	57.71 %	4.39
Apache Commons Digester	3.2	168	23.125	101	14.220	2.31	72.49 %	16.76
Apache Commons Lang	1.2	133	68.684	148	55.467	16.51	93.80 %	64.43
Apache Commons Validator	3.4	62	16.516	77	14.117	3.42	77.61 %	12.82
Joda Time	1.5.0	166	70.593	158	72.423	5.02	90.18 %	63.66

Second Order strategies were then evaluated in terms of time performance and mutation score (Judy).

All tests were run remotely on 4 x Intel Xeon 3.35 GHz CPUs, 16 GB RAM. Every test result is an average of at least 10 iterations, with code coverage computed using the Cobertura tool. The iterations of tests were launched using simple bash scripts, which also automated renaming and moving of output files into specified folders. Every run of tests was launched under a modified version of the open-source memusg.sh[1] program, which measures the average and peak memory usage of all processes invoked by the current terminal session and sub-sessions[2]. The versions of the two tools used were PITest 1.1.9 and Judy 2 (release from 13.5.2015).

Initial Performance Evaluation. The initial evaluation was run with default settings using one thread with default operators (Table 3). By default Judy has active all 56 mutation operators, while PITest 7 out of 16. Missing values in the table indicate a failure to complete the testing process.

Looking at the time performance in relation with mutation size (MS), introduced in the previous section to characterize the projects, we found a positive correlation (Spearman's Rank-Order, 0.85, p=0.0037, two-tailed). The experiments showed that with one exception, PITest is always faster to generate mutants than Judy, which generated more mutants. Similarly, when comparing how many mutants per second were generated, PITest generated mutants faster than Judy. In our experiments, Judy was not able to finish mutation analysis for larger projects (in particular Lang and Joda Time, that have the highest Mutation Size among the considered projects). When comparing time per number of mutants, Judy is generally faster for all tested projects using the default settings

[1] https://gist.github.com/netj/526585.
[2] the experimental package is available at https://goo.gl/5GPdQv.

tests. When considering the tested projects metrics, Judy is faster for smaller projects. However, for bigger projects or for projects with higher line coverage or longer tests run, the performance is rapidly lower.

Comparing average memory consumption, the same pattern applies as for comparison of tests duration. Judy consumes less memory for very small projects, but PITest shows better results for medium and bigger projects. Similarly, the peak of memory consumption is normally lower for Judy, but for big or better covered projects, the memory usage peak for Judy is a lot higher than for PITest.

Table 3. Run-time performance - default settings one thread - values in () are by using selective operators.

Project	Gen.Time (sec)		Total Time (sec)		Peak Memory (MB)	
	PITest	Judy	PITest	Judy	PITest	Judy
Commons Chain	1.1 (1.4)	3.18 (2.0)	33.5 (30.2)	5.67 (2.6)	956 (1587)	302 (240)
Commons CLI	1.3 (1.4)	12.8 (6.3)	45.8 (42.4)	228.5 (46.9)	1764 (1743)	4505 (3677)
Commons Codec	5.7 (6.2)	— (28.3)	247.6 (278.9)	— (2225.5)	3028 (3061)	— (4055)
Commons CSV	1.9 (1.6)	2.5 (1.4)	48.1 (44.1)	10.5 (4)	1648 (1654)	900 (351)
Commons DbUtils	1.6 (1.0)	6.2 (47.7)	34.2 (12)	45.6 (78.7)	784 (399)	1441 (4653)
Commons Digester	3.9 (3.0)	13.6 (8.2)	258.8 (120.1)	38.7 (20.2)	2678 (2540)	1509 (2071)
Commons Lang	21.1 (20.3)	— (—)	943.6 (907.9)	— (—)	3825 (3600)	— (—)
Commons Validator	3.5 (3.5)	13.9 (5.1)	207.9 (148.2)	135.5 (29.7)	1309 (1365)	1638 (609)
Joda Time	28.3 (28.3)	— (—)	638.8 (546)	— (—)	3857 (4267)	— (—)

*Time required for Mutation Testing is positively correlated with Mutation Size (LOCS*Coverage). It can be used as initial measure of complexity. Missing tests or tests failures (for analysis tool) hinder the possibility to apply MT.*

Mutation Operators Efficiency Results. The procedure of selection of the most efficient operators needs some further clarification. The strong mutation operators are those whose mutants are not easy to be killed. It would be very difficult to create tests that would kill 100 % of selective mutants. Therefore, we adopted a different approach by defining some thresholds to define the selective operators:

1. Run tests on all projects with all stable mutation operators (stable operators — not causing unrecoverable crashes during mutation);
2. Find most the populous (generating the highest number of mutants) mutation operators;
3. Exclude the operator if:
   - Mutation score of mutants created by the operators is higher than the average mutation score on all the tested projects;
   - The mutation operator belongs to the most populous operators and the score of mutants created is higher than the average of 80 % for all the tested projects;

Table 4. Efficiency of PITest operators

Operator	#Mut	%>avg	Operator	#Mut	% >avg
INLINE_CONSTS	12455	56	VOID_METHOD_CALLS (D)	2653	33
NEGATE_CONDITIONALS (D)	11087	100	INCREMENTS (D)	1128	100
RETURN_VALS (D)	10457	89	INVERT_NEGS	71	100
REMOVE_CONDITIONALS_EQ_IF	8335	100	REMOVE_CONDITIONALS_EQ_ELSE		
MATH (D)	3457	78	NON_VOID_METHOD_CALLS		
REMOVE_CONDITIONALS_ORD_ELSE	2752	89	CONSTRUCTOR_CALLS		
CONDITIONALS_BOUNDARY (D)	2752	22	EXPERIMENTAL_MEMBER_VARIABLE		
REMOVE_CONDITIONALS_ORD_IF	2752	67	EXPERIMENTAL_SWITCH		

Table 5. Efficiency of Judy operators

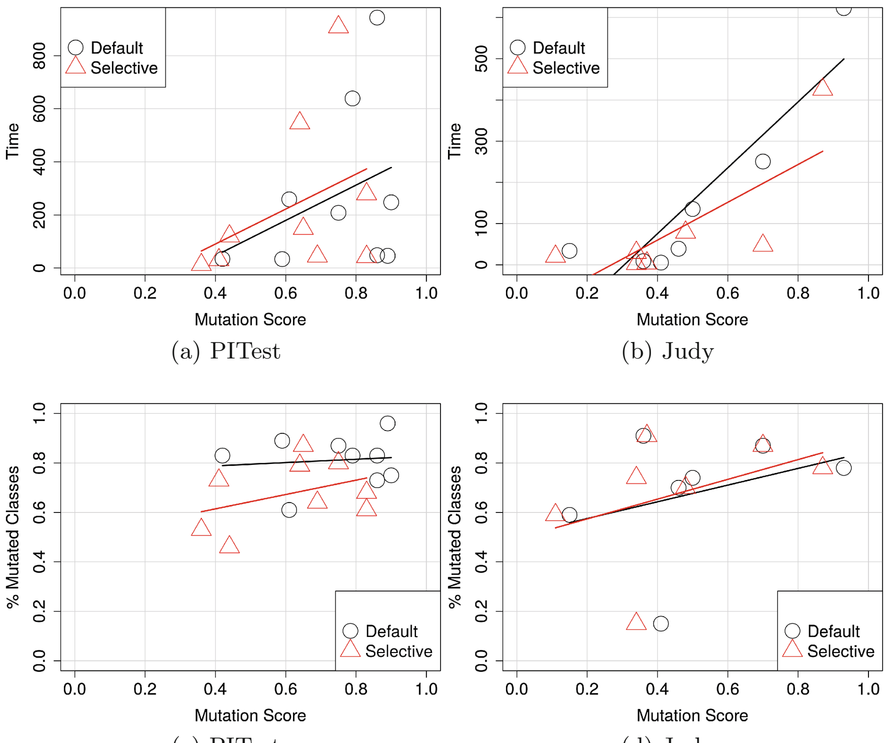
Oper.	#Mut	%>avg	Oper.	#Mut	%>avg	Oper.	#Mut	%>avg	Op.	#Mut	%>avg
JIR_Ifgt	1957	86	AIR_Mul	502	86	PNC	132	0	EOA	14	14
JIR_Iflt	1924	86	JTD	383	57	EOC	116	57	ISI	12	0
JIR_Ifle	1862	14	PRV	370	67	SIR_Ushr	98	100	JDC	12	60
JIR_Ifge	1823	86	AIR_Sub	368	86	SIR_Shl	96	100	FBD	10	50
JIR_Ifne	1575	100	EGE	269	43	JID	91	100	SCR	6	100
EAM	1391	29	PLD	259	0	SIR_Shr	76	0	IOR	5	33
JIR_Ifeq	1134	100	AIR_Add	211	67	LIR_And	58	100	CCD	0	0
OAC	1051	71	LIR_Left0	187	100	EMM	56	25	CST	0	0
JIR_Ifnull	717	100	LIR_Right	187	100	CCE	33	71	ORV	0	0
AIR_Left0	529	86	PPD	178	0	ISD	33	100	OMR	605	0
JTI	529	71	LIR_Xor	171	100	CLR	25	67	OMD	358	0
AIR_Div	525	100	IPC	165	57	DUL	23	60	IOD	206	0
AIR_Rem	510	100	LIR_Or	145	100	LSF	23	40	IOP	7	0
AIR_Right	509	86	SIR_Left0	135	100	REV	23	40	CSR	4	0

Non-excluded operators were considered selective operators and were active for the selective mutation performance tests. Tables 4 and 5 are sorted by the most populous operators from all projects (# Mut) with indication of the percentage of mutation score of the operator being higher than average mutation score ($\% > avg$)[3]. The *(D)* at the end of some operator names for PITest means that the operator is active by default. The red-painted operators are unstable ones, yellow are excluded operators and green are the selected operators for the Selective Operators experiments. The Judy operators that generated many mutants from which none were killed were considered as unstable ones.

Out of the total 16 PITest mutation operators, 5 were selected for selective mutation including the most populous operator INLINE_CONSTS, causing that the total number of generated mutants during selective mutation was almost the same as during the mutation using default PITest operators. For selective mutation using Judy, 28 out of 56 mutation operators were selected and the number of generated mutants was reduced significantly.

[3] description of operators can be found at http://pitest.org/quickstart/mutators/ and http://mutationtesting.org/judy/documentation/.

The selected operators can be used to evaluate the number of mutated classes vs the mutation score (Fig. 2a,b). The % of mutated classes refers to the number of mutated classes over the total projects' classes. The comparison of mutation score showed that the mutation score of PITest selective mutation is always lower than mutation score of default operators. This can mean that default operators are either too easy-to-be-killed, or that selected operators produced more equivalent mutants. Comparing selective vs non-selective strategies for mutation score by running a Wilcoxon Signed-Rank Test showed significant differences ($p = 0.0012 < 0.05$, two-tailed, $N = 15$).



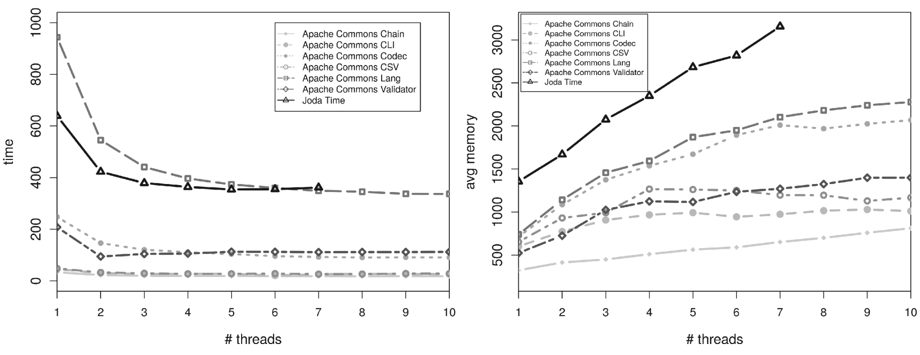
(a) PITest (b) Judy

(c) PITest (d) Judy

Fig. 2. Default vs selective mutation per mutated classes and mutation score

The total time of mutation analysis showed the real advantage of selective mutation also for PITest tests ((Fig. 2c,d and Table 3). Except of one tested project, all other were done faster using selective mutation. Comparing selective vs non-selective strategies by running a Wilcoxon Signed-Rank Test showed significant differences ($p = 0.0096 < 0.05$, two-tailed, N=16) in duration time. To note also that selective operators allowed Judy to provide results on the Apache Commons Codec project (with mutation size of 15.08).

EXP1. Using Selective Operators can bring benefits in terms of runtime performance, however, at the expense of lower mutation score. Selective Operators can also help in running Mutation Testing on some projects.



(a) Mutation Testing time vs # threads (b) Average Memory vs # threads

Fig. 3. Concurrency experiments results

Performance and Concurrency Results. The results of the concurrency experiments showed that using two or three threads can result in considerable reduction of time compared to memory consumption increase (Fig. 3). The average memory consumption is rising for all testing projects almost linearly (fitted regression up to 7 threads, *avgmem=660.42+148.55*#threads*, adj R^2=0.22), while time reduction is less than linear with the number of threads (fitted regression up to 7 threads, *time=262.81-21.71*#threads*, adj R^2=0.20).

Looking at the combined effect of decrease in time and increase in memory consumption, we considered $\Delta time$ vs $\Delta avgmemory$ (Fig. 4). In this case, time reduces less than linearly than the increase in memory (fitted regression up to 7 threads, *time=13.61-0.2656*avgmem*,adj R^2=0.45), so using more threads might increase consistently memory usage without larger benefits on time reduction.

EXP2. Up to 2–3 threads can bring high benefits in terms of runtime performance. Change in average memory consumption grows more than linearly compared to reduction in performance when increasing the number of threads.

SOM Experiments Results. We next looked at the *Higher Order Mutation Testing* strategy for the Judy project, in particular the four different algorithms (*Last2First DifferentOperators*, *ClosePair* and *RandomMix*) to combine first order mutants (FOMs) into second order mutants (SOMs) implemented in Judy [9,10]. Also in this case, we were interested in performance changes and quality of mutation score. In running the experiment, we noticed that the number

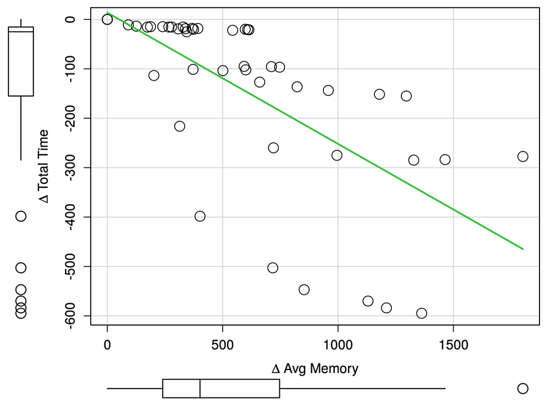


Fig. 4. Delta time vs delta average memory

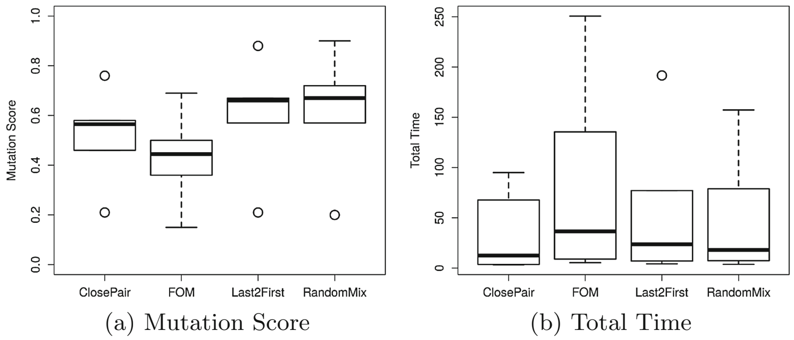


Fig. 5. Application of different SOM strategies vs FOM

of generated mutants was reduced at least by 50 % for some of the strategies and projects. Mutation score for all SOM strategies was higher than for FOM (Fig. 5a), while total time was generally lower for the three strategies in comparison with FOM (Fig. 5b). Running a Friedman non-parametric test for the differences across groups yielded significant results (0.05, two-tailed) for generated mutants ($p = 0.0089$), mutation score ($p = 0.0031$), and total time ($p = 0.0009$). However, like mentioned, the improvement of mutation score might be due to the inclusion of less equivalent mutants by applying such strategies.

When comparing individual SOM strategies, the *ClosePair* strategy gave the lowest mutation score, while *Last2First* and *RandomMix* produced very similar results for most of tested projects. This can be caused by the fact that neighboring mutants from the list of FOMs combined same type of mutants and the highest number of equivalent SOM mutants were generated using this strategy.

EXP3. SOM strategies improve the results in terms of mutation score and in terms of generated mutants, having positive benefits on the performance. However, manual inspection is needed to understand how many equivalent mutants are generated.

Threats to Validity. We have several threats to validity to report [17].

For *internal validity*, measurements performed were averaged over several runs to reduce the impact of external concurring for resources. One of the main issues is the reliability of mutation score as quality indicator. The score always includes equivalent mutants as the automated detection is undecidable [9], and the only way to discover them is by manual inspection — unfeasible for larger projects. In fact, two projects with the same mutation score might be quite different depending on the number of equivalent mutants. We also used thresholds for the definition of the selective operators, and some sensitivity analysis can be more appropriate to define the best ranges.

Related to *external validity*, we cannot ensure that results generalize to other projects. However, we selected 9 heterogeneous projects in terms of size, code coverage. More insights will be given by testing on even larger software projects of industrial partners. Furthermore, the package of the current experiments will be available to increase external validity by means of replications.

For *conclusion validity*, we applied several statistical tests and simple linear regression in different parts of the experiment. We always used the non-parametric version of the tests, without normality distribution assumption, and we believe to have met other assumptions (type of variables, relationships among measurements) to apply each test.

4 Related Works

There are several related works about experimental evaluations of tools for automated mutation testing in JVM environments.

One of the first experimental evaluations [13] was done on the mutation operators of the Mothra project omitting two, four and six of the most populous mutation operators. The test cases killed 100 % of mutants generated by selective mutation. These test cases were then run on non-selective mutants. These test cases killed almost 100 % percent of mutants. Out of 22 mutation operators used by Mothra, 5 were key operators that provide almost the same coverage as non-selective mutation [14].

Madeyski et al. provided an experimental evaluation comparing the performance of generating mutants between Judy and MuJava on various Apache Commons libraries [10]. From the experiments, Judy was able to generate at least ten time more mutants per second as MuJava.

In 2011, the applicability of mutation testing was examined in Nica et al. [11]. The selected tools were MuJava, Jumble and Javalanche, focusing on the performance of generating mutants. The only tool able to generate mutants was MuJava generating about 123 class-level mutants and 30,947 method level mutants in approx. 6 hours. Jumble and Javalanche showed configuration difficulties and low performance. The main conclusion was that mutation testing was too slow to be used in real world software projects.

In 2013, Delahaye et al. compared Javalanche, Judy, Jumble, MAJOR and PITest on several sample projects [4]. The results showed that Jumble, MAJOR and PITest were able to finish the analysis for every project, while Judy generated the highest number of mutants and Javalanche the lowest number of mutants. The research indicated that mutation testing tools still need a lot of improvements to be usable in real world projects.

In 2015, Rani et al. compared MuJava, Judy, Jumble, Jester and PITest in an experimental evaluation [15]. The experiments were run on set of short programs (17-111 LOC). The research showed that all the tools produced almost the same average mutation scores except of PITest, which produced 25 % higher score than the rest of the tools. One of the conclusions was that a new mutation system for Java should be created, with faster generation and execution of mutants.

In 2016 Klischies et al. run an experimentation considering PITest on several Apache Commons projects. As metric for the experiments authors use the inverse of mutation score, as an indication of the goodness of the mutation operator set. They overall considered Mutation Testing applicable to real world projects with a low number of equivalent mutants, inspected manually, on the set of projects that were considered. However, strong concerns remained for the applicability to larger projects and in case code coverage within projects is too low, making the whole mutation analysis less effective [8].

Our work is different from the aforementioned set of related works as we focus on the selection of the best mutation operators and mutation strategies for improvements in performance on a set of medium sized projects. We can directly compare the SOM experiments results with Madeyski et al. [9], getting the same results in terms of increase of mutation score and performance improvement.

5 Discussion

There are several findings about the application of Mutation Testing that we put forward in the current paper. The general performance of Mutation Testing is impacted by Mutation Size, that is the size of the project and the code coverage level (*RQ1*). When taking into account the applicability of Mutation Testing, is appropriate to consider Mutation Size (LOCs size and code coverage) as an indication of the time required. This can be a good indicator to use by analogy for the application to other projects. A good strategy for the application of Mutation Testing is, in fact, to first increase code coverage to good levels, as having lower code coverage levels cannot tell much about the quality of tests. Clearly, larger coverage impacts on the execution of the tests, while mutant generation and mutant compilations stay the same. Taking into account multiple threads, time reduction decreases less than linearly with the increase of memory consumption. Mutation Testing can be optimized by looking at points in which parallelization does not bring enough incremental benefits. For the set of projects considered, 2–3 threads are effective numbers for performance / memory resources optimization.

Identifying the most efficient operators and applying *selective operators* improves the results in terms of runtime performance at the expense of lower mutation score and lower number of mutated classes (*RQ2*). This is a strategy that can be applied to extend the applicability of Mutation Testing to allow to run the approach to wider set of projects. In the selective strategy we looked at the efficiency of the operators in terms of killed mutants, but other approaches may look at the operators that generate more mutants. Based on the results, we believe that this set of strategies can help to apply Mutation Testing within industrial contexts, as default configurations can lead to a larger overhead in running the process. However, practitioners would need to fine-tune the Mutation Testing environments according to the specific projects needs. We included a list of selected operators efficiency based on the overall set of projects, that can give indications for application to other projects.

We looked at the impact of *Second Order Mutation* to recombine *First Order Mutants* and reduce in this way the number of mutants *RQ3.* All different sub-strategies considered (*Last2First DifferentOperators*, *ClosePair*, *RandomMix*) improve in terms of time required to run mutation testing, with higher mutation score than considering the initial mutants. However, while improvements in time are due to the lower number of generated mutants, mutation score can be influenced by equivalent mutants, as such manual inspection would be suggested to look for the effect on each considered project.

6 Conclusion

Mutation Testing is still an evolving testing methodology that can bring great benefits to software development. With increasing computational resources, it can reach wider adoption within industry, aiding to build more robust software. However, there are still aspects that hinder its usage, namely the computational complexity, equivalent mutants and possible lack of integration tools [9].

In this paper, we looked at the current support of Mutation Testing in JVM environments, with an experimental evaluation based on industrial partner's needs. We focused on various aspects of performance, evaluating different strategies that can be applied to reduce the time needed for mutation analysis. We evaluated how *selective operators* and *second order mutants* can be beneficial for the mutation testing process, allowing to reduce runtime overhead. Based on the results, we believe that Mutation Testing is mature enough to be more widely adopted. In our case, the experimental results have been useful for knowledge transfer in an industrial cooperation, with future works aimed at exploring the experimented approaches on the company's source code repositories.

Acknowledgments. We are grateful to the developers of both PITest and Judy for feedback provided in the usage of the tools. In case of Judy, the SOM experiments have been possible with a newer version provided by the developers.

References

1. CJBS Insight: Cambridge university study states software bugs cost economy $312 billion per year. http://insight.jbs.cam.ac.uk/2013/financial-content-cambridge-university-study-states-software-bugs-cost-economy-312-billion-per-year/
2. Crispin, L., Gregory, J.: Agile Testing: A Practical Guide for Testers and Agile Teams. Pearson Education, Boston (2009)
3. Dedík, V., Rossi, B.: Automated bug triaging in an industrial context. In: 42nd EUROMICRO Conference on Software Engineering and Advanced Applications, pp. 363–367. IEEE (2016)
4. Delahaye, M., Du Bousquet, L.: A comparison of mutation analysis tools for Java. In: 13th International Conference on Quality Software (QSIC), pp. 187–195. IEEE (2013)
5. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: help for the practicing programmer. Comput. **11**(4), 34–41 (1978)
6. Jia, Y., Harman, M.: Higher order mutation testing. Inf. Softw. Technol. **51**(10), 1379–1393 (2009)
7. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011)
8. Klischies, D., Fögen, K.: An analysis of current mutation testing techniques applied to real world examples. In: Full-scale Software Engineering/Current Trends in Release Engineering, p. 13 (2016)
9. Madeyski, L., Orzeszyna, W., Torkar, R., Jozala, M.: Overcoming the equivalent mutant problem: a systematic literature review and a comparative experiment of second order mutation. IEEE Trans. Softw. Eng. **40**(1), 23–42 (2014)
10. Madeyski, L., Radyk, N.: Judy-a mutation testing tool for Java. IET Softw. **4**(1), 32–42 (2010)
11. Nica, S., Ramler, R., Wotawa, F.: Is mutation testing scalable for real-world software projects. In: VALID Third International Conference on Advances in System Testing and Validation Lifecycle, Barcelona, Spain (2011)
12. Offutt, A.J., Lee, A., Rothermel, G., Untch, R.H., Zapf, C.: An experimental determination of sufficient mutant operators. ACM Trans. Softw. Eng. Methodol. **5**(2), 99–118 (1996)
13. Offutt, A.J., Rothermel, G., Zapf, C.: An experimental evaluation of selective mutation. In: Proceedings of the 15th International Conference on Software Engineering ICSE 1993, pp. 100–107. IEEE Computer Society Press, Los Alamitos (1993)
14. Offutt, A.J., Untch, R.H.: Mutation 2000: uniting the orthogonal. In: Wong, W.E. (ed.) Mutation Testing for the New Century, pp. 34–44. Kluwer Academic Publishers, Norwell (2001)
15. Rani, S., Suri, B., Khatri, S.K.: Experimental comparison of automated mutation testing tools for Java. In: 2015 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO), pp. 1–6. IEEE (2015)
16. Roy, N.K.S., Rossi, B.: Towards an improvement of bug severity classification. In: 40th EUROMICRO Conference on Software Engineering and Advanced Applications, pp. 269–276. IEEE (2014)
17. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering. Springer, Heidelberg (2012)