An Experimental Evaluation of PIT's Mutation Operators

Michael Andersson

Michael Andersson
VT 2017
Bachelor thesis, 15 credits
Supervisor: Pedher Johansson
Examiner: Lars-Erik Janlert
Bachelor of Science Programme in Computing Science, 180 credits

Abstract

Mutation testing is a fault-finding software testing technique that creates mutants by injecting a syntactic change into the source code. This technique operates by evaluating the ability of a test suite to detect mutants, which is used to asses the quality of a test suite. The information from mutation testing can be used to improve the quality of a test suite by developing additional test cases. It is a promising technique, but the expensive nature of mutation testing has prevented its widespread, practical use. One approach to make mutation testing more efficient is selective mutation testing. This approach proposes to use a subset of the available mutation operators, thereby reducing the number of mutants generated which will lead to a reduced execution time. According to theory many mutants are redundant and do not provide any useful information in the development of additional test cases. This study will evaluate PIT, a mutation testing tool, by using selective mutation testing on five programs, and compare the effectiveness of PIT with MuJava. The results showed that PIT's default operators generated a small and effective set of mutants and at the same time giving a satisfying mutation score. However, there was no significant difference in mutation score produced by the different sets of operators. The test suites adequate for PIT managed to detect 75-85% of MuJava's mutants, which is relatively good when taken into consideration that PIT generated less than half as many mutants.

Acknowledgements

I want to express my gratitude to my supervisor Pedher Johansson, for the advice you have provided during this project. I would also like to thank my friends and family for all support. Special thanks towards my little brother Anton, for all the math tutoring he has given me over the years.

Contents

1 Introduction

Software testing is a practice to ensure the quality of the software. Test cases are used to examine the behaviour of an application and to ensure that the application behaves accordingly. Researchers and software engineers use test criteria to measure the quality of test cases; one criterion that has been around for more than forty years is mutation testing.

Mutation testing operates by evaluating the ability of a test suite to distinguish between the original application and a set of altered versions, called mutants. A mutant is created by injecting a syntactic change into the original program, which can lead to a semantic fault. One of the main hypothesis behind mutation testing is its ability to represents real-world faults; this ability can be used by software testers to design test cases[2].

Mutation testing is a powerful and promising technique, but it is not without problems. One problem is the high computational cost coming with executing a large number of mutants against a test suite [6]. Besides the computational cost there exists equivalent mutants. These mutants can not be detected by any test case, and have to be manually marked and excluded from the final result [6]. Several approaches have been proposed [6] in an attempt to reduce the cost of mutation testing. One approach has been to reduce the number of generated mutants without gaining a significant loss of test efficiency. An other approach has focused on reducing the computational cost by optimizing the execution process. This thesis will focus on selective mutation testing, which is an approach to reduce the number of generated mutants.

In selective mutation testing, the number of mutants are reduced by using a subset of the available mutation operators without gaining a significant loss of test effectiveness relative to non-selective mutation testing (all operators). The idea is that many of the generated mutants are redundant and does not provide any useful information on the design of additional test cases. So, finding a small subset of efficient operators, which produces useful mutants that creates strong test cases, are important for making mutation testing practical.

The mutation testing tool Pitest(PIT) [3] will be the subject of this study. The author(s) of PIT [3] claims that some operators are more prone in producing equivalent and redundant mutants, thus this set is deactivated by default. It would be interesting to investigate if their assumption is correct, or if there exists an alternate set of operators that is more efficient. Since PIT is a relatively new tool the effectiveness of the test suites, adequate for operators in PIT, will be measured in how sufficient they are in detecting mutants generated by MuJava, a comprehensive mutation testing tool [9]. This thesis will use selective mutation to evaluate PIT's operators and try to find subset of operators that can produce results comparable to non-selective mutation testing, and measure how effective PIT is compared with MuJava.

1.1 Mutation testing

Mutation testing was first introduced 1978 by Richard A. Demillo [5]. Mutant testing is based on two hypotheses: The first one is called the *competent programmer hypothesis* [10]. It states that a competent programmer writes programs with simple faults, which can be fixed by making a single change to the source code. The second hypothesis is called the *coupling effect* [10]. It asserts that small, simple faults can cause a snowball effect, and cascade into more complex faults. The traditional process of mutation analysis is illustrated in Figure 1, and normally consists of three steps: Generating mutants, executing the test suite, and analyse the outcome. After the final step, the tester can use the information given from the analysis to design additional test cases to strengthen the test suite and improve the mutation score.

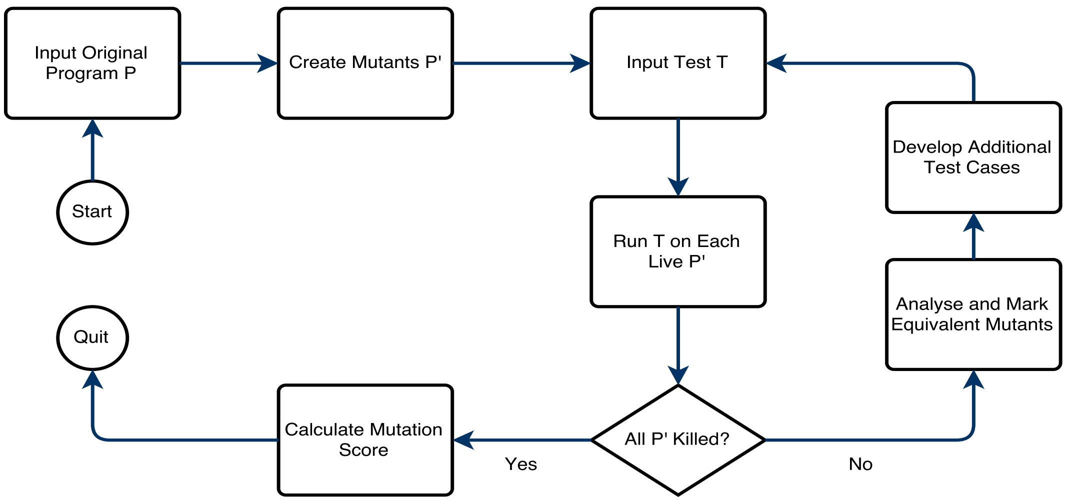

Figure 1: Traditional process of mutation analysis

The first step of mutation analysis [13] is to generate mutants of the original program. This is done by inducing small syntactic changes according to well formed rules called mutation operators. Mutants are created by applying an operator on the source code; thus, every mutant will contain a single syntactic change that differs from the original program. Mutants are often created with an automatic testing tool and operates either on bytecode or source code. If mutants are generated from the source code the program have to be recompiled for each created mutant, which is computationally expensive. However, when using bytecode translation all operators can not be represented, but this problem is outweighed by the the reduced execution time [6].

When the mutants have been generated, a test suite is executed against the original program and the mutants to compare the output from the test suite. If the output differs from the original program, then the mutant is said to be detected or killed, otherwise it is said to have survived [6]. If any mutants remained undetected, the surviving mutants can by killed through mutant analysis, which can help developing additional test cases. Mutation testing tools normally provide some sort of information about the mutants that can help the augmentation of the test suite.

After test case execution the surviving mutants need to be analysed. This is either done to mark equivalent mutants since they need to be omitted from the final mutation score, or to use the information to develop additional test cases. The end result will produce a mutation score, which is the ratio of the number of mutants killed over the total number of non-equivalent mutants. The goal of mutation analysis is to raise the mutation score close to 1, i.e. 100% [10].

1.2 Mutant classification

During the mutation analysis mutants can be classified into different categories. Redundant mutants [7] are a type of mutant which can be killed by almost every test case, therefore not contributing any useful test cases to the test suite, and omitting operators that produces high numbers of redundant mutants will reduce execution time. In contrast to redundant mutants there exists *Hard-to-kill mutants* [8]. These mutants are particular interesting because they lead to strong test cases since they are only killed by a small percentage of test cases. The theory is that redundant mutants subsumes hard-to-kill mutants, thus one can omit operators that tend produces redundant mutants without a significant loss in the quality of the final test suite.

Some mutants behave equivalent to the original program, and therefore not representing a fault. Mutants of this type are called equivalent mutants. For example a mutation operator might change the condition in a for-loop, like below:

```
for (int i = 0; i < 10 ; i++) {        for (int i = 0; i != 10 ; i++) {
        ...                                    ...
}                                      }
```

In the example above, the code has changed, but the behaviour of the for-loop is the same, which makes the mutant equivalent since it can not be detected by any test case. A mutant might also be unkillable due to constraints or limitations in a programming language. The effort of detect equivalent mutants can be high, even for small programs, causing unnecessary system processing, and the software engineer have to spend energy on a non-existent problem. The detection of equivalent mutants is a hindrance for practical usage of mutation testing.

1.3 Cost reduction techniques

Mutation testing is an expensive testing technique, and several approaches have been made to reduce the cost. One way is to optimize the process of generating mutants; by mutating byte-code instead of source code will eliminate the need to recompile the program for every mutant. Other approaches have focused on reducing the number of generated mutants.

Mutant sampling is a method were all possible mutants are generated with the full set of mutant operators, but only a certain x% of the generated mutants are used in the analysis [6]. This method can serve as a cost-effective alternative to applications which do not demand ultra-high mutation score; since it only reduce the effectiveness with a small percentage [15].

Selective mutation is not random as mutation sampling, but instead aims to reduce the number of mutants by omitting certain operators without a significant loss of test efficiency [6]. One approach is to omit operators that produces most mutants, since many of these mutants might be redundant and killed too easily, and instead use operators that produces hard-to-kill mutants. Since hard-to-kill mutants require test cases of higher quality which will lead to a stronger test suite. Other approaches have categorized operators based on the syntactic elements they modify. A study [11] comparing selective mutation with non-selective mutation (all operators) showed that expression modification operators, which replaces operators and insert new ones, achieved 99.5 % mutation score. Their results indicates that selective mutation testing can drastically reduce the execution time and make mutation testing more effective.

1.4 PIT: Mutation testing framework

The process of seeding faults into a program is done by an automatic mutation testing tool. A mutation testing tool are normally designed for a specific programming language. Some of the older mutation tools have been developed mainly for research purposes and not for mainstream use, but several languages have some support for mutation testing, for example C, C#, SQL and Java [6]. This thesis focus on tools for Java.

PIT [4] is an open source mutation testing framework for Java, it was developed for practical usage and is well integrated with modern development tools. In short, PIT generates mutants by byte code translation, and stores them in RAM memory which minimizes memory overhead. PIT only executes test cases that will reach the mutated code, thus avoiding unnecessary executions. PIT presents line coverage and mutation score from the mutation analysis in HTML-format. The HTML-report also shows a copy of the code, displaying where the mutants were created, and how many mutants that were killed/survived, thus facilitates the development of additional test cases and the analysis of equivalent mutant.

PIT employs 13 official mutation operators, which are presented in Appendix A. Seven of them are activated by default. These operators are designed to generate hard-to-kill mutants, and generating a minimal amount of equivalent mutants. The other six operators are deactivated by default since they are more prone in creating equivalent mutants and deemed not as effective [3]. PIT employs a rather small set of mutation operators compared to other tools, the reason have been to reduce the execution time by limiting the number of generated mutants [4].

MuJava [9] is an other tool developed for the Java language. MuJava employs 15 operators that conforms with the recommendation made in literature, and also generates mutants via byte code translation. MuJava has been developed for and by academics, and has been widely used in research projects [6] [14].

1.5 Previous research

One major obstacle for mutation testing is the high computational cost that comes with the large amount of mutants even a small program can generate. Research have been conducted to find a way to make mutation testing more efficient by lowering the number of generated mutants by selectively chose mutation operators. Offutt et al. [11] made an empirical study where they compared selective mutation testing with standard, non-selective mutation testing (all operators). They discovered that 5 operators produced a mutation score of 99.5% when measured against non-selective mutation. Their result showed that selective mutation is almost as strong as non-selective mutation with almost a four-fold reduction in the number of mutants.

Smith and Williams [14] made a similar study but evaluated operators in MuJava. Their study were conducted on a web application and came to the conclusion that some mutation operators where more useful than others, but they could not find any significant difference between operators. They believed that the choice of mutation operators should reflect on the type of application that is being tested.

Ammann et.al [1] wanted to find a theoretical way to estimate how many mutants that are needed and presented a theoretical framework for mutant set minimization. They tested their model on a particular test set and their approach focuses mainly on reducing the number of redundant mutants. The presented model could efficiently create a minimum set of mutants with the same result as non-selective mutation. Even though their results seems promising, their model requires an existing test suite with high mutation score and selective chose which mutation operators that produces subsuming mutants. Since this problem is undecidable, it impossible to know beforehand which minimal set to use.

Even though selective mutation testing have shown promise in relatively small programs there have been doubt if it suits large, real-world programs. Zhang et.al [16] wanted to investigate this issue. In their experiments they strived for a sufficient mutation score of at least 90% and they found that selective mutation testing scaled well. The number of mutants in selective mutation testing increases slowly as the size of the program increases. This shows that selective mutation testing is promising for larger programs.

Laurent et al. [8] noticed that the popular mutation analysis tool PIT employed a restrictive set of mutant operators that did not fully conform recommendations made by literature. This can be problematic, since the effectiveness of mutation analysis depends highly on which mutation operators that are being used. Even though PIT has a lower execution time than other tools, it should not be at the expense of the method effectiveness. They presented an extended version of PIT with additional mutation operators, which are more in line with the sufficient set of operators presented by Offutt et al [11]. They conducted experiments comparing the original operators and the extended operators in terms of effectiveness, easiness and scalability. The experiment showed that the mutants generated by the extended operators were considerably harder to kill, and encoded faults not captured by the original operators, which indicates the the operators present in PIT have room for improvements.

2 Experiment

The focus of this study is to evaluate PIT's operators by using selective mutation testing. This will be done by developing test suites adequate for each set of operators (achieving a mutation score of 100% when executed against a given set), and measure how adequate each test suite is in detecting mutants generated by non-selective mutation (all operators). The strength and quality of a set is measured in the test suite that comes from it. To get a better grasp of the overall quality of PIT, an experiment will be conducted to measure how adequate the test suites are in detecting mutants generated by MuJava. The interesting data will be the mutation score each test suite manages to produce when executed against the non-selective mutation operators; the number of mutants and test cases will also be taken into consideration when assessing the effectiveness of a set.

2.1 Subject Programs

The programs that are going to be used in the experiments are presented in Table 1. The same programs have been used before in experiments on mutation testing [6]. The source code for the programs were gathered from *Introduction to Software Testing* by Jeff Offutt and Paul Ammann [1]. The book provided a link where the source code from the book can be downloaded.

Table 1 Programs used in the experiments

Java programs	Lines of Code	Methods	Description.
Quadratic	80	4	Calculates the roots of a quadratic equation.
TestPattern	60	1	Pattern matching of strings.
TrashAndTakeOut	35	2	Does arithmetic operations based on input value
TriType	70	2	Calculates triangle type
Cal	40	1	Calculates the number of days between two dates

2.2 Division of mutation operators

The operators provided in PIT were divided into five sets. Two sets are activated and deactivated by default and are presented in Appendix A. Three more sets were constructed based on the syntactic elements they modify, see Table 2. *Replacement*

Table 2 Shows how the mutation operators are divided based on the syntactic elements they modify.

Replacement of Operand	**Expression Modification**	**Statement Modification**
Inline Constants	Conditional Boundary	Return Values
Expr. Member Variables	Negate Conditionals	Non Void Method Calls
Expr. Switch Mutator	Math	Void Method Calls
Remove Conditionals	Invert Negatives	Constructor Calls

of operand operators replaces each operand with another legal operand, *Expression Modification* operators replaces expression operators with another, *Statement Modification* operators modifies entire statements.

2.3 Experimental Procedure

The first step was to generate test suites for each set of operators. I created all test cases by hand in the JUnit testing framework, and additional test cases were generated by analysing the surviving mutants. Only test cases that were responsible for killing at least one mutant were added into the final test suite. The process of test case generation continued until all non-equivalent mutants were killed and all equivalent mutants have been marked.

When all the necessary test cases, for each set of operators, had been created, each test suite were executed against non-selective mutation and a mutation score was calculated.

$$E(M_S,T_s) = \frac{M_k(T_s)}{M_{ne}} * 100\% \tag{2.1}$$

Equation 2.1 was used for this purpose, which measures how well a set of operators compares against non-selective mutation testing. M_s represents the subset of operators under evaluation, T_s represents the test suite constructed to kill the mutants generated by M_s. $M_k(T_s)$ represents the number of mutants killed by T_s when non-selective mutation was used. M_{ne} is the number of non equivalent mutants generated by non-selective mutation.

3 Results

This section will start by analysing data from the test suites adequate for PIT's operator, and their adequacy in detecting non-selective mutants. The section concludes by analysing on how sufficient the test suites, adequate for PIT's operators, were in detecting mutants generated by MuJava. All experiments were performed on an Dual-core Intel I7-7560U with 16GB of RAM and running Ubuntu 16.04.2 LTS 64-bit operative system. PIT were run through a Maven plug-in.

3.1 Results from PIT

The number of mutants reflects upon the execution time, and therefore plays an important role when assessing the efficiency of operators. As seen in Figure 2, the two sets: *Deactivated by default* and *Replacement of operand*, produces on average the highest number of mutants. Some sets only had a small reduction in the number of mutants that were generated for some of the programs, which indicates that the number of mutants depends on the inner structure of the program, and what possibilities there exist the operators to mutate the code. There existed no significant difference in the number of equivalent mutants produced by each set.

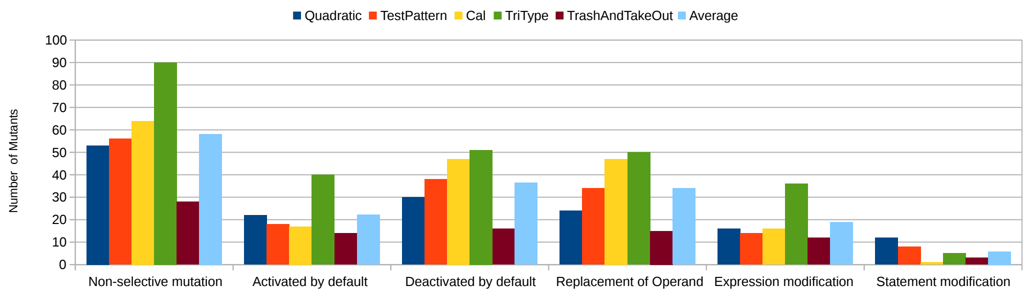

Figure 2: The number of mutants each set of operators produced for every program, along with the results on how many mutant that were generated by non-selective mutation. Results were collected from the experiments on PIT.

The number of mutants seems to reflect upon how many test cases that were required to make a test suite adequate (achieves 100% mutation score for a specific set). Figure 3 shows how many test cases that were required by each set. It is especially clear with the sets that generated the lowest number of mutants, like the set *Statement modification* whose test package required the lowest number of test cases to be adequate. This pattern also exists with the two sets that produced most mutants: *Deactivated by default* and *Replacement of operand.* As seen in Figure 3, these two sets required, on average, the highest number of test cases to create adequate test suites.

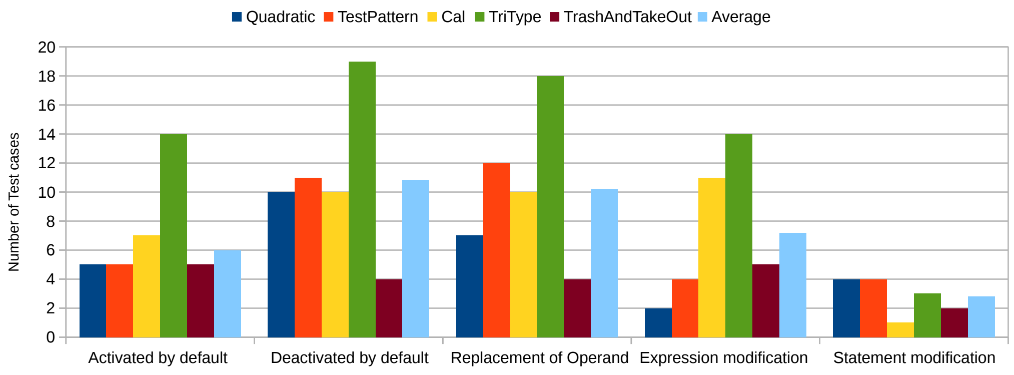

Figure 3: The number of test cases required for each set of mutation operators to create an adequate test suite.

The results seems to indicate that it requires a certain amount of mutants and test cases to achieve a satisfying mutation score. As the set *Statement modification*, seen in Figure 4, produced the lowest number of mutants and test cases, and were achieving the lowest mutation score against non-selective mutation. Similar pattern exists for the two sets: *Deactivated by default* and *Replacement of operand*, who generated the highest number of mutants and test cases achieved the highest mutation score against non-selective mutation. These two sets produced a stable result with a mutation scores that gave small variations among the programs compared to the other sets. However, the two sets *Activated by default* and *Expression modification* were achieving similar mutation score and at the same time generating fewer mutants, and required fewer test cases to be adequate. It can be argued, because of these reasons that *Activated by default* and *Expression modification* are slightly more efficient since they require less manual work to achieve a good result.

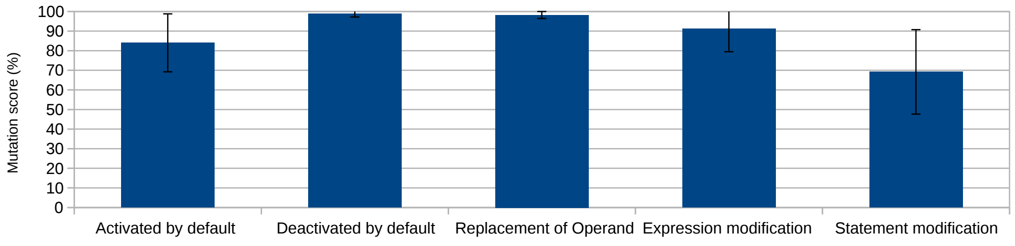

Figure 4: The mutation score that each test suite, adequate for a specific set of mutation operators, managed to produce against non-selective mutation operators (all operators). Calculations were made with Equation 2.1, presented in section 2.3.

3.2 Comparison to MuJava

The different test suites, adequate for PIT's operators, were also measured in their ability in detecting mutants generated by MuJava. MuJava will in this sense be used as a benchmark, and as a reference point to measure the strength of PIT's operators. The results from this experiment are shown in Figure 5, along with a test suite that were adequate in detecting all mutants generated by PIT.

The test suites developed for PIT managed to kill 75-85% of Mujava's mutants. The result follows a similar pattern as the previous experiment, given that *Deactivated by default* and *Replacement of operand* produced, on average, the highest and most stable mutation score, and *Statement modification* produced the worst mutation score. This further strengthens the conclusion that a certain amount of test cases might be required to achieve high mutation score. The test suite adequate for detecting all of PIT's mutants was able to detect around 85-90% of MuJava's mutants. The results may even be higher due to the fact that some equivalent mutants may not have been marked and excluded from the final mutation score.

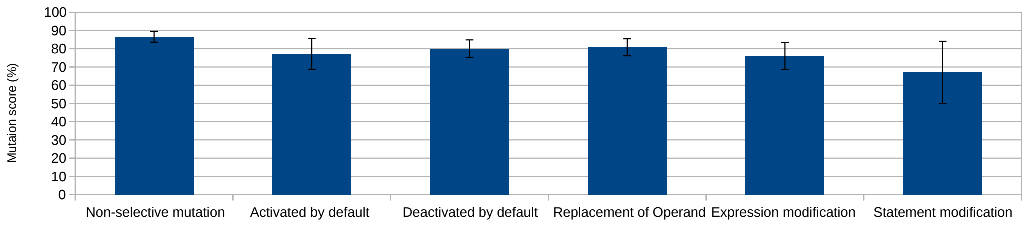

Figure 5: The mutation score each test suite, adequate for a set of operators, was able to produce against the traditional mutation operators used by MuJava. For this experiment an additional test suite, that were adequate for all operators in PIT, were developed and used to measure the overall quality of PIT.

The test suites created for PIT managed to kill a high percentage of MuJava's mutants, which shows that the operators in PIT generates hard-to-kill mutants that has lead to strong test cases. This is supported by Figure 6 which shows that PIT produces far less mutants than MuJava and strengthens the claim that PIT's operators produce higher quality mutants, that is, more difficult to detect than MuJava's mutants. MuJava generates more than double the amount of mutants than PIT, which suggests that a large number of MuJava's mutants are redundant, and that PIT employs an efficient set of operators that values quality over quantity.

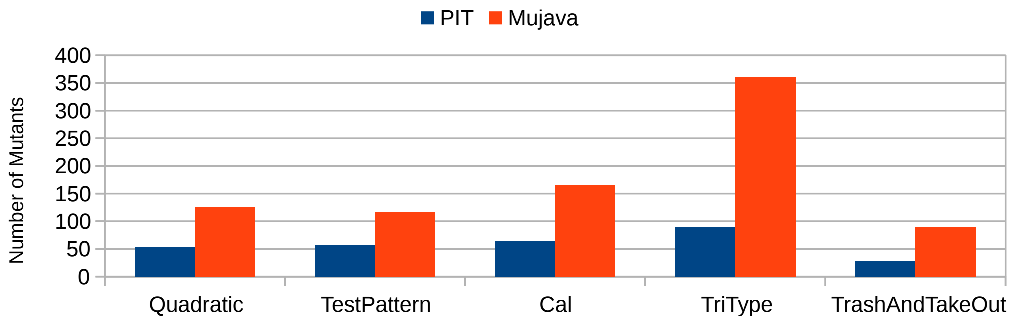

Figure 6: Shows a comparison in the number of mutants produced for each program by MuJava and PIT.

4 Discussion

This thesis has evaluated the mutation operators used by PIT, and tried to determine the most efficient set. Besides from mutation score, the number of mutants and test cases played an important role when assessing the effectiveness of a set. Since a large number of mutants will lead to increased execution time, and require extensive manual labour to analyse and might lead to an excess of weak, and unnecessary test cases.

Due to inconclusive results only *Statement modification* operators can be ruled as inefficient; the set generated few mutants and test cases, which may be the reason why it did not achieve the same results as the other sets. Two of the sets: *Deactivated by default* and *Replacement of operand* that generated most mutants and test cases, generated a high mutation score but not significantly higher than the sets *Activated by default* and *Expression modification.* The latter sets generated on average the lowest number of mutants and test cases, and at the same time offered the best reduction of mutants compared with non-selective mutation testing (all operators). Similar but more definite results were produced by Offutt [11], they used a mutation testing tool that generated far more mutants than PIT, which might be the reason for their more decisive result. It may also be that a breakpoint has been reached, were you can not reduce the number of mutants without affecting the efficiency of mutation testing.

One study [12] have compared mutation testing tools for Java and notice that PIT generated far less mutants compared with other tools. This raised a concern about quality of the test data that were adequate for PIT, and if PIT were effective enough in comparison with other mutation testing tools. The test suites performed relatively good when they were executed against MuJava's mutants, and managed to detect 75-85% of its mutants. The results are not overwhelming but when taken into consideration that PIT generated less than half the number of mutants than that of MuJava, the results becomes more impressive. This shows that PIT employs an effective set of operators that generates a small but effective set of mutants.

Conducting the experiments with more programs would have likely lead to similar results. In order to gain a better understanding of the effectiveness of mutation operators and mutation testing in general, future experiments should be conducted on larger scale programs to determine what aspects that need to change or improve to make mutation testing more viable for companies.

5 Conclusion

This thesis has shown that PIT compared relatively well with MuJava. PIT generated less than half the number of mutants than that of MuJava and only achieved 10-25% drop in test suite effectiveness. Taken this into consideration, and how well integrated PIT is with modern development tools, and how easy PIT is to set-up compared to MuJava, shows that PIT is on the right track of making mutation testing a viable testing method. If you are going to use PIT as your mutation testing tool, I would recommend using the default operators. They managed to generate a restrictive set of useful mutants and at the same time giving satisfying results.

References

[1] Paul Ammann, Marcio Eduardo Delamaro, and Jeff Offutt. Establishing theoretical minimal sets of mutants. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, pages 21–30, Washington, DC, USA, 2014. IEEE Computer Society.

[2] Paul Ammann and Jeff Offutt. *Introduction to Software Testing.* Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[3] Henry Coles. PIT Mutation Testing mutation testing tool for java and the jvm. `http://pitest.org/`, 2015. Accessed: 2016-03-28.

[4] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. Pit: A practical mutation testing tool for java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 449–452, New York, NY, USA, 2016. ACM.

[5] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[6] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.*, 37(5):649–678, September 2011.

[7] Bob Kurtz, Paul Ammann, Jeff Offutt, and Mariet Kurtz. Are we there yet? how redundant and equivalent mutants affect determination of test completeness. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 142–151, April 2016.

[8] Thomas Laurent, Anthony Ventresque, Mike Papadakis, Christopher Henard, and Yves Le Traon. Assessing and improving the mutation testing practice of PIT. *CoRR*, abs/1601.02351, 2016.

[9] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. Mujava: A mutation system for java. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 827–830, New York, NY, USA, 2006. ACM.

[10] A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, January 1992.

[11] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, April 1996.

[12] Shweta Rani, Bharti Suri, and Sunil Kumar Khatri. Experimental comparison of automated mutation testing tools for java. In *2015 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions)*, pages 1–6, Sept 2015.

[13] Pedro Reales, Macario Polo, Jose Luis Fernández-Alemán, Ambrosio Toval, and Mario Piattini. Mutation testing. *IEEE Software*, 31(3):30–35, May 2014.

[14] Ben H. Smith and Laurie Williams. An empirical evaluation of the mujava mutation operators. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, pages 193–202, Sept 2007.

[15] Eric W. Wong and Aditya P. Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185 – 196, 1995.

[16] Jie Zhang, Muyao Zhu, Dan Hao, and Lu Zhang. An empirical study on the scalability of selective mutation testing. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 277–287, Nov 2014.

A PIT's mutation operators

Table A1: The mutation operators in PIT which are activated by default, and illustrates what elements of the code they modify along with an example. [4]

Name	**Transformation**	**Example**
Conditional Boundary	Replaces one relational operator instance with another	$< \rightsquigarrow \leq$
Increments	Replace increment with decrement, and vice versa	$-- \rightsquigarrow ++$
Invert Negatives	Removes negative from a variable	$-a \rightsquigarrow a$
Math	Replaces a numerical operation by another one	$+ \rightsquigarrow -$
Negate Conditionals	Negates one relational operator	$== \rightsquigarrow != $
Return values	Transforms the return value of a function	return 0 $\rightsquigarrow$ return 1
Void Method calls mutator	Deletes a call to a void method	$voidfunc(a) \rightsquigarrow$

Table A2: The mutation operators in PIT that are deactivated by default, and illustrates what elements of the code they modify along with an example. [4]

Name	**Transformation**	**Example**
Constructor call	Replaces constructor calls by null	$newC() \rightsquigarrow null$
Inline constant	Replaces a constant by another one or increments it.	$1 \rightsquigarrow 0$, $a \rightsquigarrow a+1$
Non void Method call	Deletes a call to a non-void method	$intdigit(c) \rightsquigarrow$
Remove Conditionals	Replaces a condition branch with true or false	$if(...) \rightsquigarrow if(true)$
Member Variable	Replaces an assignment to a variable with the Java default	$a = 5 \rightsquigarrow a$
Switch	Replaces switch statement labels by the Java default	$if(...) \rightsquigarrow if(true)$