This is a preprint of an article published in Lech Madeyski, Norbert Radyk: "Judy - a mutation testing tool for Java", IET Software, 2010, vol. 4, iss. 1, pp. 32–42.
http://dx.doi.org/10.1049/iet-sen.2008.0038

Judy – a mutation testing tool for Java

Lech Madeyski, Norbert Radyk
Institute of Informatics, Wrocław University of Technology,
Wyb.Wyspiańskiego 27, 50370 Wrocław, POLAND
Lech.Madeyski@pwr.wroc.pl, Norbert.Radyk@gmail.com

Abstract

Popular code coverage measures, such as branch coverage, are indicators of the thoroughness rather than the fault detection capability of test suites. Mutation testing is a fault-based technique that measures the effectiveness of test suites for fault localization. Unfortunately, use of mutation testing in the software industry is rare because generating and running vast numbers of mutants against the test cases is time-consuming and difficult to do without an automated, fast and reliable tool. Our objective is to present an innovative approach to mutation testing that takes advantage of a novel aspect-oriented programming mechanism, called "pointcut and advice", to avoid multiple compilation of mutants and, therefore, to speed up mutation testing. An empirical comparison of the performance of the developed tool, called Judy, with the MuJava mutation testing tool on 24 open-source projects demonstrates the value of the presented approach. The results show that there is a statistically significant ($t(23) = -12.28$, $p < .0005$, effect size $d = 3.43$) difference in the number of mutants generated per second between MuJava ($M = 4.15$, $SD = 1.42$) and Judy ($M = 52.05$, $SD = 19.69$). Apart from being statistically significant, this effect is considered very large and, therefore, represents a substantive finding. This therefore allows us to estimate the fault detection effectiveness of test suites of much larger systems.

1 Introduction

Mutation testing is a fault-based technique that measures the effectiveness of test suites for fault localization, based on seeded faults [1, 2]. Mutation testing may be used to judge the effectiveness of a test set, as the test set should kill all the mutants. The fault detection effectiveness of a test suite is defined as the percentage of faults that can be detected by that test suite. A mutation is a change made to the source code by a mutation testing tool. Faults are introduced into the program by creating a collection of faulty versions, called mutants. These mutants are created from the original program by applying mutation operators which describe syntactic changes to the original code. The

test suite is then used to execute these mutants and to measure how well it is able to find faults. A test case that distinguishes (returning a different result) the program from one or more of its mutants is considered to be effective in finding faults [3].

A mutation score is a quantitative measurement of the quality of the test suite and is defined as a ratio of the number of killed mutants to the total number of non-equivalent mutants [4]. The total number of non-equivalent mutants results from the difference between the total number of mutants and the number of equivalent mutants. Equivalent mutants always produce the same output as the original program, so they cannot be killed. Unfortunately, determining which mutant programs are equivalent to the original program is an extremely tedious and error-prone activity. One of the practical solutions, suggested by Offutt and Untch [5], is to ignore equivalent mutants. As a result, Madeyski [6, 7] proposed the use of a mutant score indicator which is a lower bound on the mutation score obtained by ignoring equivalent mutants.

The total mutation testing time is equal to the sum of the mutants generation time and the mutants execution time [8]. We assume that the mutants execution time is unaffected by the mutation testing tool being used because the impact of the extra method calls due to the advice turned out to be negligible. Hence, the empirical comparison of the performance of mutation testing can be based on mutant generation times only. A measure of performance of the mutant generation process, used in Section 6, is the number of mutants generated (and compiled, in the case of source code manipulation) per second ($NMPS$). The exception is the performance comparison of the tools on academic projects in Section 2 where both the mutant generation and execution times are taken into account. This is due to the fact that the academic projects are considerably smaller than the open-source projects used in Section 6.

Section 2 starts with the presentation of the related work and presents the main activities in the field of mutation testing, while Section 3 includes a comparison of mutation testing tools.

A new approach to mutation testing is presented in Section 4, while Section 5 introduces Judy [9] – a new mutation testing tool for Java. Judy supports the phases of generation, compilation to bytecode, and execution (i.e. testing) of mutants. It is developed in Java and AspectJ [10] to test Java programs, it implements both traditional and object-oriented (OO) mutation operators, and it supports JUnit [11, 12] (the *de facto* standard among unit testing frameworks [13, 14, 15, 16]) and Ant [17] (a Java build tool) to help practitioners carry out mutation testing.

In the presented approach, we create (dependent on the mutation operators enabled) a collection of mutants of the system under test. Meta-mutants are used to manage the collection of mutants and to avoid multiple (i.e. time-consuming) compilations for every mutant. This was made possible with the help of pointcut and advice Aspect-Oriented Programming (AOP) mechanism. As a result, the speed of mutant generation is improved. An empirical evaluation of the speed of mutant generation for open-source projects is presented in Section 6, while a summary of the paper is presented in Section 7.

2 Related work

Research in mutation testing has a rich history [5] and focuses on four kinds of activities [18]: 1) defining mutation operators, 2) experimentation, 3) developing tools, 4) reducing the cost of mutation analysis. The first one involves defining new mutation operators for different languages and types of testing [3, 18, 19, 20, 21].

The second research activity is experimentation with mutations [22, 23, 24]. Empirical studies have supported the effectiveness of mutation testing. Mutation testing has been found to be more powerful than statement and branch coverage [25] and more effective in finding faults than data flow [26, 27]. Offutt et al. [22, 28] and Wong and Mathur [29] evaluated the idea of selective mutation which identifies the critical mutation operators that provide almost the same testing coverage as non-selective mutation. Using this approach considerably decreases the number of mutants generated and thus reduces computational cost. Offutt et al. [22] suggested just five sufficient operators (ABS, AOR, LCR, ROR, UOI) for Fortran. Andrews et al. [23] suggest that mutants can provide a good indication of the fault detection ability of a test suite. Andrews et al. [24] used mutation analysis for the assessment and comparison of testing coverage criteria.

The third kind of activities in mutation testing research is developing mutation tools. Mothra [30] and Proteum [31] were developed for Fortran and C, respectively. Jester [32], Jumble [33], MuJava [8, 34, 35], JavaMut [36] and Response Injection (RI) [37], as well as the tool presented in this paper, are dedicated to the Java language. A comparison of mutation testing tools for Java is presented in Section 3.

Last but not least in terms of research activities is investigating ways to reduce the cost of mutation analysis. The major cost of mutation analysis arises from the computational expense of generating and running large numbers of mutant programs. Mutation testing is a powerful but time-consuming technique which is impractical to use without a reliable, fast and automated tool that generates mutants, runs the mutants against a suite of tests, and reports the mutation score of the test suite. The approaches to reduce this computational expense of generating and running large numbers of mutant programs usually follow one of three strategies [5]: 1) "do fewer": seek ways of running fewer mutant programs with minimal information loss, 2) "do smarter": seek to distribute the computational expense over several machines or to avoid complete execution, 3) "do faster": focus on the ways of generating or running each mutant program as fast as possible. Offutt et al. [34] have implemented three versions of MuJava using different implementation techniques: source code modification, bytecode modification (i.e. instrumentation of the bytecode of compiled Java programs), and mutant schema (i.e. encoding all mutants for a program into one "meta-mutant" program, which is subsequently compiled and run at speeds considerably higher than achieved by interpretive systems [38]). Bogacki and Walter's [37] approach, as well as the approach presented in this paper, both fall into this last category of research activities.

3 Comparison of existing tools

An important characteristic of mutation testing tools are the mutation operators supported by a tool. Two types of mutation operators for object-oriented (OO) languages can be distinguished, as suggested by Ma et al. [39]: (1) traditional mutation operators adapted from procedural languages and (2) OO (or class) mutation operators developed to handle specific features in OO programming.

Other essential characteristics of mutation testing tools are mutant generation methods (via bytecode or source code modification) and speedup techniques (e.g., mutant schemata generation (MSG)) [38]. The MSG method is used to generate one "meta-mutant" program at the source level that incorporates many mutants [8].

A comparison of the facilities offered by Java mutation testing tools is provided in Table 1 along with some of their advantages and disadvantages.

Table 1: Comparison of mutation testing tools for Java

Characteristic	MuJava [8, 34, 35]	Jester [32]	Jumble [33]	RI [37]	Judy
Traditional, selective mutation operators (ABS, AOR,LCR,ROR,UOI)	Yes	No	No[1]	No[2]	Yes
OO mutation operators	Yes	No	No	No	Yes[3]
Mutant generation level	bytecode[4]	source code	bytecode	–	source code
Produces non-executable mutants	Yes[5]	Yes	Yes[4]	–	Yes
Meta-mutant	Yes[4]	No	No	Yes[6]	Yes
Mutants format	separate class files	separate source files	in-memory	–	grouped in source files
JUnit support	No	Yes	Yes	Yes	Yes
Build tool or command line support	No	Yes	Yes	–	Yes

[1] The selective operators are not provided. They could be implemented, but not easily, by modifying the code of the tool.

[2] The RI prototype uses only simple mutation operators, i.e. changing primitive types and the objects of the String class.

[3] Only on the intra-method level.

[4] According to Offutt et al. [34], three versions of MuJava have been implemented using different implementation techniques: source code modification, bytecode modification, and mutant schema. There were some problems with the two latter techniques [34] but, finally, they were combined [8]. This means that some mutations are still done with source code analysis using the mutant schema.

[5] Although generation mechanisms based on bytecode transformation reduce the chance of creating non-executable mutants in comparison to the ones based on source code transformation, they do not ensure that all mutants created using this technique will execute (i.e. changing a visibility modifier from `public` to `private` (AMC mutation operator [41]) for a method invoked externally in other project packages will create a non-executable mutant, but this can be done using bytecode transformation).

[6] Meta-mutant includes only mutants for a given method [37].

3.1 MuJava

According to Ma et al. [8], JavaMut [36] is not freely available for download, does not support the execution of mutants and separately compiles each mutated class (which is a very slow process). Therefore, MuJava started with Chevalley and Thévenod-Fosse's work [36] with the goal of addressing the remaining problems [8]. As a result, MuJava can be seen as a successor of JavaMut and is included in the comparison presented in Table 1. MuJava [8, 34, 35] offers a large set of both traditional and OO mutation operators dedicated to the Java language (a comprehensive description of the implemented mutation operators is presented in [40] and [41]). Moreover, it implements two advanced techniques, bytecode transformations and MSG, with the aim of reducing the mutant generation time. The MSG/bytecode translation method implemented in MuJava is reported to be about five times faster than the separate compilation approach [8]. MuJava, however, has some limitations that may prevent practitioners from using this tool in everyday development. For example, it does not provide a command line interface (which prevents running in batch mode) and does not work with JUnit. However, the recently released MuClipse plugin [42], acting as a bridge between the MuJava mutation engine and the Eclipse IDE, seems to overcome this limitation. In fact, MuClipse exhibits some limitations concerning the automation of mutants generation. For example, it is not possible to adjust the test classes naming convention of MuClipse to the naming convention used by the system under test (SUT) or to test easily the whole project or a single package. Also there are limitations with the testing process (e.g., it is not possible to select more than one class simultaneously, and it is necessary to select additional tests for each class explicitly). As a result, mutation testing using MuClipse and MuJava is not easy to run in a production environment. Moreover, lack of integration with a build tool (Ant or Maven) is still an issue. In spite of those limitations, MuJava is worth detailed empirical comparison.

3.2 Jester

Jester [32] is a tool that was found to be useful (but too slow) by some practitioners (e.g., Fowler in an interview with Venners [43]). However, Offutt argued that Jester turned out to be a very expensive way to apply branch testing, rather than mutation testing, owing to an oversimplified mechanism of mutants generation [44]. Actually, Jester offers a way to extend the default set of mutation operations, but problems concerning performance and reliability of the tool (see below), as well as a limited range of possible mutation operators (based only on string substitution) remain. Moreover, the mutation operators offered by Jester are not context-aware, and often lead to broken code. It is worth mentioning that Jester's approach is to generate, compile and run unit tests against a mutant. The process (shown in Figure 6) repeats for every mutant of the SUT and, thus, is inefficient. Because of these major disadvantages, Jester was excluded

from detailed empirical evaluation.

3.3 Jumble

Jumble operates directly at a bytecode level to speed up mutation testing, it supports JUnit and it has been deployed in an industrial setting [33]. It was therefore worthy of further investigation. Unfortunately, the limited set of mutation operators supported by Jumble required us to disable most of the Judy mutation operators to enable comparisons. The mutation operators supported by Jumble are, in fact, simplified versions of AOR, ASR, COR, LOR, ROR and SOR studied in [45]. They are simplified in such a way that only one of the mutations defined by the mutation operators is, in fact, applied (e.g., '–' is replaced by '+', and not by each of the other operators, i.e. '+', '*', '/', and '%', as the AOR mutation operator assumes). Including Jumble in the empirical comparison of the tools would require a drastic reduction of the set of mutation operators, i.e. excluding OO mutation operators. Furthermore, Jumble does not support any means of adjusting the naming convention of unit tests. As different conventions are used in open-source projects, bringing Jumble in the comparison would require a huge amount of additional effort.

Judy has been compared to Jumble in a set of academic projects to answer the question whether mutation operators implemented in Judy and Jumble return similar results. It turned out that Pearson's correlation coefficient of the reported mutation score indicator results for Judy and Jumble is positive and very high, $r = 0.89$, while Cronbach's alpha coefficient, an indicator of internal consistency, is even higher (0.94). Hence, we may conclude that both tools are measuring the same underlying construct.

3.4 Response Injection (RI)

RI is a prototype that takes advantage of aspect-oriented mechanisms to generate mutants. It uses only simple mutation operators, i.e. changing primitive types and the objects of the String class. Unlike MuJava and Judy, the only mutation applied to objects is null value. Furthermore, it is not clear how RI would support the wide range of mutation operators designed for Java, as the mutation testing approach was simplified to check only the result of a method, exceptions thrown or changes to fields of a class. Another limitation is that the RI approach does not allow the testing of changes in objects passed via method parameters. The documentation, as well as the code of the prototype, is not available. Hence, RI is not included in further empirical comparison.

3.5 Summary

Judy will be described in detail in Sections 4 and 5. The result of this short comparison shows that available mutation testing tools for Java have very different characteristics. On the basis of the characteristics presented in Table 1, MuJava seems to have several important advantages (meta-mutants and bytecode transformation) that can make the tool very efficient. Furthermore, MuJava is the only tool (except Judy) that supports OO mutation operators. The comparison of Judy and MuJava required the implementation of a dedicated MuJava client to automate both the mutant generation and compilation phases. As mentioned by the MuJava authors, the most serious barrier to the practical use of mutation testing is the unacceptable computational expense of generating vast numbers of mutants [5, 39].

4 FAMTA Light – A new approach to mutation testing

FAMTA Light (fast aspect-oriented mutation testing algorithm) is a novel approach to mutation testing that was inspired by the need for an efficient and reliable mutation testing tool for use on real-world projects. This approach takes advantage of the pointcut and advice mechanism, which allows the definition of special points in the program flow (during method execution, constructor invocation and class field value change) – called pointcuts – and related methods – called advices – that can be invoked before, after or instead of a defined pointcut. The main strength of the mechanism is related to its support for regular expressions, what enables the definition of similar behaviour for specific types of event. This introduces very effective solutions to some commonly known programming issues, called "cross-cutting concerns", such as logging or transaction management – see [10, 46] for more details.

A detailed analysis of Jester revealed that the most time-consuming part of the mutation testing process is the mutant compilation phase repeated for each mutant. Therefore we looked for a way to reduce the number of compilations and thus reduce the time spent in the compilation phase. The approach, called FAMTA Light, is presented in Figure 1, while the main phases of the algorithm are described in Sections 4.1, 4.2 and 4.3.

4.1 Mutant generation process

In comparison with existing mutation testing solutions, FAMTA Light introduces some important changes in the mutant generation process. The basic assumption is that, during the generation process, we aim to create, on the

basis of the enabled mutation operators, a collection of mutants of the SUT (in the form of additional methods placed in mutated classes) instead of a single mutant. Additionally, meta-mutants are generated to manage the collection and to avoid multiple compilations for every mutant.

The pointcut and advice mechanism is used in the meta-mutant generation process. In the FAMTA Light testing phase, every meta-mutant is responsible for managing the collection of mutants related to one of the classes of the tested system. It contains a set of class-related advices and pointcuts. Each advice is responsible for the execution of one method of the tested system class and is invoked instead of the method due to a defined pointcut. In the advice body a decision is made either to invoke the original method or one of its mutations. This limits the possible set of supported mutation operators to the intra-method ones, i.e. the mutated code remains within a single method body. In order to introduce the inter-method, intra-class and inter-class mutation operators, it is indispensable to employ a different mutants generation technique (e.g. bytecode modifications).

The use of meta-mutants makes it possible to perform mutant generation by changing the source code of the SUT. However, instead of a single mutant, a collection of mutants is generated. The mutations are generated in the form of mutant methods, being modifications of the original methods. The generation process defined in this way enables meta-mutants to dynamically invoke the mutant methods instead of the original ones, and the behaviour depends on the internal state of the mutation tester. The mutation tester state is represented by the value of a static `counter` field and handled by a `switch` instruction of a meta-mutant aspect.

We explain the presented approach with an example using Java and AspectJ syntax [10] (see Figure 2). The example consists of a single class `ClassA`, with a single method `isEven`, that is responsible for checking whether the number passed as a parameter is even. For the class `ClassA`, mutations of the original `isEven` method are generated in such a way that they extend the original class with new methods `isEven_mutation0` and `isEven_mutation1` as shown in Figure 3. Also generated is the meta-mutant aspect (see `ClassAAspect` in Figure 4), which is used to manage the mutants collection.

4.2 Mutant compilation process

The generation process presented in Section 4.1 provides the single mutant compilation solution, where mutants already created during the generation process are compiled at the same time and there is no need to repeat the compilation for every generated mutant. In consequence, a huge speedup during the mutant compilation phase is observed, which influences the overall mutation testing time. A detailed empirical evaluation of Judy is given in Section 6.

4.3 Mutants testing process

The FAMTA Light mutants testing process is the phase during which all the created and compiled mutants are tested. It is the key phase of the mutation testing process during which, for each tested class, the test suite is run against the dynamically activated mutants. A pseudo-code description of the mutants testing phase is given in Figure 5.

Mutants are activated for the tested class on the basis of internal mutation tester state (the value of the `counter` attribute of the meta-mutant aspect – see Figure 4). The `counter` attribute is modified at the end of each mutant test in order to activate the next mutant. The testing process ends when all tested system mutants have been activated.

It is worth mentioning that in the FAMTA Light algorithm, every new mutant is an additional, modified copy of a mutated method, not a whole system or class. Therefore, the algorithm copes quite well when the number of packages, classes, methods and mutation operators is large.

5 Judy

Judy is an implementation of the FAMTA Light approach developed in Java with AspectJ extensions. The core features of Judy are high mutation testing process performance (see Section 6), advanced mutant generation mechanism (Judy supports 16 predefined mutation operators presented in Table 2), integration with professional development environment tools, full automation of mutation testing process (which makes the tool useful in automated build environments common in industrial and open-source projects) and support for the latest version of Java (supported Java versions are: 1.3, 1.4, 5.0, 6.0), enabling it to run mutation testing against the most recent Java software systems or components.

Judy, like MuJava, supports traditional mutation operators (ABS, AOR, LCR, ROR and UOI). These were initially defined for procedural programs and have been identified by Offutt et al. [22] as selective mutation operators. These operators are to minimize the number of mutation operators, whilst maximizing testing strength. The latter was measured by Offutt and Lee [22] by computing the nonselective mutation scores of the test sets that were 100% adequate for selective mutation. Other mutation operators dedicated to the Java language and supported by Judy are UOD, SOR, LOR, COR, and ASR (see Ammann and Offut [45]), along with the aforementioned selective operators. Judy supports the EOA and EOC mutation operators proposed by Ma et al. [3] that can model object-oriented (OO) faults, which are difficult to detect [39].

EAM, EMM, JTD and JTI mutation operators [3] were added, because there was no reason to exclude them and there is still no standarized set of selective mutation operators for class mutation operators [39]. Judy can be extended to support more mutation operators.

Table 2: Mutation operators supported by Judy

Abbreviation	Description	Example mutation	Reference
ABS	Absolute value insertion	$x = 2 * a; \rightarrow x = 2 * abs(a);$	[21]
AOR	Arithmetic operator replacement	$x = a + b; \rightarrow x = a * b;$	[21]
LCR	Logical connector replacement	$x = a \&\& b \rightarrow x = a \| \| b$	[21]
ROR	Relational operator replacement	$if(a > b) \rightarrow if(a < b)$	[21]
UOI	Unary operator insertion	$x = 2 * a; \rightarrow x = 2 * -a;$	[21]
UOD	Unary operator deletion	$if(a < -b) \rightarrow if(a < b)$	[45]
SOR	Shift operator replacement	$x = a << b; \rightarrow x = a >> b;$	[45]
LOR	Logical operator replacement	$x = a \& b; \rightarrow x = a \| b;$	[45]
COR	Conditional operator replacement	$if(a \&\& b) \rightarrow if(a \& b)$	[45]
ASR	Assignment operator replacement	$x+ = 2; \rightarrow x- = 2;$	[45]
EOA	Reference assignment and content assignment replacement	$List\ l1, l2;\ l1 = new\ List();$ $l1 = l2; \rightarrow l1 = l2.clone()$	[3]
EOC	Reference comparison and content comparison replacement	$Integer\ a = new\ Integer(1);$ $Integer\ b = new\ Integer(1);$ $boolean\ x = (a == b); \rightarrow$ $boolean\ x = (a.equals(b));$	[3]
JTD	`this` keyword deletion	$this.r = r; \rightarrow r = r$	[3]
JTI	`this` keyword insertion	$this.r = r; \rightarrow this.r = this.r$	[3]
EAM	Accessor method change	$circle.getX(); \rightarrow circle.getY();$	[3]
EMM	Modifier method change	$circle.setX(1); \rightarrow circle.setY(1);$	[3]

5.1 Implementation issues

Unfortunately constraints imposed by the Java development environment meant that it was not possible to implement Judy (based on FAMTA Light approach) exactly as planned. The major reason for this was the size of the generated mutated classes which could run to tens of thousands of lines, causing severe performance and reliability issues in compilation (e.g. throwing OutOfMemoryError).

We therefore changed the initial FAMTA Light single iteration approach (generation, compilation, and then the testing phase) and introduced an iterative solution, which constrained the maximum number of mutations that could be applied to a single Java class within the single iteration ($MAXMUT$). As a result, all of the mutants are generated but if the number of mutants exceeds $MAXMUT$ they are generated in subsequent iterations (i.e. $MAXMUT$ constraint applies only to a single iteration). The total number of iterations in the mutation testing process has changed to $\left\lfloor \frac{NMUT}{MAXMUT} \right\rfloor + 1$, where $NMUT$ is the highest number of mutations generated for a single class among all the SUT classes. The configurable $MAXMUT$ value can be adjusted in order to tune the mutation testing performance to the project characteristics. For most of the tested projects, the best performance was observed for the $MAXMUT$ value between 50 and 100.

A second problem was related to the mass compilation of mutants where a compilation error caused by any of the generated mutants results in breaking all simultaneously compiled mutants. Therefore a compilation error handling mechanism was introduced. If a compilation error occurs then the mutant that

caused the error is removed from the mutant set which, in a further attempt, gets recompiled.

Disappointingly, this resulted in repeated compilations that had a negative impact on the overall mutation testing performance. To minimize this impact, the mutant generation mechanism was supplemented by dedicated syntactic and semantic rules to help to identify valid mutants. These rules were derived from the analysis of a large set of cases which led to compilation errors during the testing of open-source projects (i.e. ASF projects, Hibernate, Eclipse). Their introduction significantly reduced the influence of repeated compilations on the overall Judy performance.

6 Empirical evaluation on open-source projects

The aim of this section is to evaluate the performance of Judy. As mentioned in Section 1, performance of the mutant generation process is defined as the number of mutants generated per second. Since MuJava implements a large set of mutation operators as well as complex solutions (MSG, bytecode modification) to increase the overall mutation testing process performance [8, 35], we will use it for evaluating tools. In order to increase the validity of the empirical evaluation, real-world open-source Java projects from the Apache Software Foundation (ASF) are used.

Because of the limitations of MuJava mentioned in Section 3, we must restrict our comparison of MuJava and Judy to the generation and compilation phases. Furthermore, it is difficult to separate both phases as MuJava operates on bytecode transformations. Our approach is justified in the case of both tools, as these phases lead to the generation of the set of already compiled mutants. Also, the overall time of the testing phase should be similar, as it mainly depends on the test suite execution time.

6.1 Experiment description

The following definition determines the foundation for the empirical study [47]:
Object of study: The objects of study are mutation testing tools.
Purpose: The purpose is to evaluate the impact of the selected mutation testing tool (MuJava or Judy) on the mutation testing process conducted on software development products.
Quality focus: The performance of the mutant generation process, measured by the number of executable mutants generated per second.
Perspective: The perspective is from the researcher's point of view.
Context: The study is run on 24 real-world open-source software products selected at random from the Apache Software Foundation's projects written in Java which commonly take advantage of JUnit tests.

The independent variable is the mutation testing tool for Java – MuJava or Judy. The dependent (response) variable is the number of mutants generated per second ($NMPS$). In order to measure the mutants generation time for MuJava, we implemented a dedicated MuJava to automate the mutant generation and compilation phases for MuJava. The experiment was conducted on the same set of mutation operators (AOR, ROR, COR, SOR, LOR, EOA, EOC, EAM and EMM). The following null hypothesis is to be tested: $H_{0\ NMPS,\ MuJava/Judy}$ – there is no difference in the number of mutants generated per second between MuJava and Judy. The ability to generalize from the context of the experiment is further elaborated when discussing threats to the validity of the study.

The design of the study is one factor (the mutation testing tool), with two treatments (MuJava and Judy). The design resulted in a balanced design, i.e. design in which all treatments have equal sample size – 24 open-source products.

6.2 Analysis of the experiment

The experimental data, presented in Table 3, were analysed with descriptive analysis and statistical tests. Descriptive statistics for $NMPS$ are shown in Table 4. The box plot in Figure 7 presents the shape of the distribution of the results. Although visual inspection of the Figure 7 suggests that there were in fact performance differences, with Judy performing better, statistical tests will be conducted to answer the question whether the impact of mutation testing tool on $NMPS$ is significant, or not.

A common misconception is that a statistically significant result is always of practical significance, or demonstrates a large effect in the population. Given a sufficiently large sample, even small differences can be found to be statistically significant but the number of analysed open-source projects is not so large as to fall into that trap. Practical significance looks at whether the difference is large enough to be of value in a practical sense. Therefore, not only statistical significance, but also the effect size and the 95% confidence interval are reported. Unlike significance tests, effect size is independent of sample size and shows how much more effective the new tool is. The large effect size indicates that the difference between the tools is substantial and meaningful beyond simple statistical significance.

The exploratory analysis of the collected data has shown that the assumptions of the parametric tests are met. Therefore, the hypothesis from Section 6.1 is evaluated by means of the dependent t-test. The t-test was conducted to evaluate the impact of the mutation testing tool on $NMPS$. The results ($t(23) = -12.28$, $p < .0005$) show that there is a statistically significant difference in $NMPS$ between MuJava ($M = 4.15$, $SD = 1.42$) and Judy ($M = 52.05$, $SD = 19.69$). The effect size statistic $d = (M_1 - M_2)/\sqrt{(SD_1^2 + SD_2^2)/2} = 3.43$ where M_i and SD_i are the mean and standard deviation for each treatment. This is considered to be a very large effect size. The mean difference (in $NMPS$) between Judy and MuJava is -47.90 and the 95% confidence interval for the estimated mean difference is between -39.83 and -55.97.

Table 3: Number of mutants per second on open-source projects

Project	$NMPS$		Lines	Number of	Branch
	MuJava	Judy	of code	test cases	coverage
Apache Jakarta Commons Beanutils	4.43	52.64	18244	424	50%
Apache Jakarta Commons CLI	5.77	52.45	4121	108	69%
Apache Jakarta Commons Chain	2.06	42.47	7753	116	55%
Apache Jakarta Commons Codec	3.50	68.28	4699	189	83%
Apache Jakarta Commons DBCP	2.49	72.00	11291	274	51%
Apache Jakarta Commons DbUtils	1.73	23.21	3221	82	57%
Apache Jakarta Commons Digester	1.39	40.75	12885	159	35%
Apache Jakarta Commons Discovery	3.20	30.29	5000	11	29%
Apache Jakarta Commons Email	5.21	36.82	3177	75	35%
Apache Jakarta Commons FileUpload	5.52	38.78	5300	31	72%
Apache Jakarta Commons IO	5.96	70.60	14326	477	19%
Apache Jakarta Commons JXPath	4.43	53.99	28197	443	41%
Apache Jakarta Commons Lang	3.24	79.99	48290	1850	—[1]
Apache Jakarta Commons Launcher	3.18	31.39	3364	0	0%
Apache Jakarta Commons Logging	3.04	33.44	5997	109	27%
Apache Jakarta Commons Math	5.68	99.32	42212	1247	95%
Apache Jakarta Commons Modeler	4.94	56.91	7981	7	0%
Apache Jakarta Commons Transaction	4.05	43.03	6622	59	57%
Apache Jakarta Commons Validator	5.46	82.33	9906	277	43%
Apache Torque	4.70	57.38	22577	51	9%
Apache Velocity	3.43	25.80	44037	129	28%
Apache Velocity Tools	6.61	69.02	16593	21	—[1]
Apache XML Security Java	4.42	48.24	44304	373	—[1]
Apache XMLBeans	5.16	40.00	84804	0	0

[1]Branch coverage results were obtained using Cobertura 1.9. During testing of Apache Jakarta Commons Lang, Apache Velocity Tools and Apache XML Security, Java errors occurred and no results were returned.

6.3 Validity evaluation

When conducting the experiment, there is always a set of threats to the validity of the results [47]. Hence, the possible threats are enumerated.

Threats to the statistical conclusion validity are considered to be under control. Robust statistical techniques, tools (SPSS) and enough sample sizes to assure statistical power are used. Furthermore, calculating $NMPS$ does not involve human judgement.

Regarding internal validity, the experiment was conducted on projects selected at random. However, it turned out that MuJava was not able to finish the analysis of some larger projects (e.g., Apache Xerces Java XML Parser) due to memory related issues (`OutOfMemoryError`).

Threats to the construct validity are considered not very harmful. Using a single type of measure carries a mono-method bias threat. The performance measure ($NMPS$) seems to be an accurate reflection of the performance concept. However, there is a threat of the experimenters' expectancies, since we are also the inventors of the presented approach and tool. Only an independent replication would remove this threat.

Table 4: Descriptive statistics for the number of mutants per second ($NMPS$)

Measure	Mutation testing tool	Mean (M)	Max	Median (Mdn)	Min
Number of mutants per second ($NMPS$)	MuJava	4.15	6.61	4.43	1.39
	Judy	52.05	99.32	50.35	23.21

The threat to external validity stems from the fact that the analysed projects were open-source projects, not industrial ones. However, nowadays, open-source projects are often used as the building blocks of industrial applications, in a wide range of application domains. In addition, open-source projects are often big and developed by large teams of professional developers around the world. Some indications of the similarities between open-source projects and industrial ones are also given by Rothfuss [48]. He pointed out several differences (concerning different resource needs, voluntary participation, high level of software reuse etc.) but these differences are not considered very harmful.

7 Summary

Mutation testing is not meant as a replacement for code coverage, but as a complementary approach that is useful in detecting those pieces of the code that are executed by running tests, but are not actually fully tested. It is not widely used in software engineering due to the limited performance of the existing tools [5, 18, 39] and the lack of support for standard unit testing and build tools.

Our approach uses aspect-oriented mechanisms to increase the performance of the mutation testing process, while providing a framework that can be extended to support additional intra-method mutation operators. We have shown that, when applied to analysed projects, the new tool, Judy, outperforms MuJava. However, extending the range of mutation operators to inter-method, intra-class and inter-class operators would require combining meta-mutant and source code manipulation techniques implemented in Judy with other mechanisms, for example bytecode modification technique, as mentioned in Section 4.1. Therefore, the tool combining strengths of the aforementioned techniques and offering an even wider set of mutation operators is under active development.

The use of the FAMTA Light algorithm eliminates the huge overhead of repeated generation and compilation phases and, therefore, leads to a situation where total mutation testing time mainly depends on the execution time of the test suite. Further improvements in mutation testing performance may be made by decreasing the number of mutations (e.g., by means of selective mutations for Java) or the time of the test suite execution.

Acknowledgements

This work has been financially supported by the Ministry of Education and Science as a research grant 3 T11C 061 30. Judy is short for Judyta – the name of the youngest daughter of the first author.

References

[1] DeMillo, R.A., Lipton, R.J., and Sayward, F.G.: 'Hints on Test Data Selection: Help for the Practicing Programmer', IEEE Computer, 1978, 11, (4), pp. 34–41

[2] Hamlet, R.G.: 'Testing Programs with the Aid of a Compiler', IEEE Trans.Softw. Eng. 1977, 3, (4), pp. 279–290

[3] Ma, Y.S., Kwon, Y.R., and Offutt, J.: 'Inter-Class Mutation Operators for Java'. Proc. 13th Int. Symposium Software Reliability Engineering, Washington, USA, November 2002, pp. 352–363

[4] Zhu, H., Hall, Patrick.A.V., and May, J.H.R.: 'Software Unit Test Coverage and Adequacy'. ACM Comput. Surv. 1997, 29, (4), pp. 366–427

[5] Offutt, A.J., and Untch, R.H.: 'Mutation 2000: Uniting the Orthogonal', in Wong, W. E. (Ed.): 'Mutation testing for the new century' (Kluwer Academic Publishers, 2001, 1st edn.), pp. 34–44

[6] Madeyski, L.: 'On the Effects of Pair Programming on Thoroughness and Fault-Finding Effectiveness of Unit Tests'. Proc. Int. Conf. Product Focused Software Process, Riga, Latvia, July 2007, pp. 207–221, http://dx.doi.org/10.1007/978-3-540-73460-4_20

[7] Madeyski, L.: 'The Impact of Pair Programming on Thoroughness and Fault Detection Effectiveness of Unit Tests Suites', Softw. Process Improve. Pract., 2008, 13, (3), pp.281–295, http://dx.doi.org/10.1002/spip.382

[8] Ma, Y., Offutt, J., Kwon, Y.R.: 'MuJava: an automated class mutation system', Softw Test Verif Rel, 2005, 15, (2), pp. 97-133

[9] Judy, http://www.e-informatyka.pl/sens/Wiki.jsp?page=Projects.Judy, accessed June 2007

[10] Laddad, R.: 'AspectJ in Action: Practical Aspect-Oriented Programming' (Manning Publications Co., 2003, 1st edn.)

[11] Massol, V., and Husted, T.: 'JUnit in Action' (Manning Publications Co., 2003, 1st edn.)

[12] Rainsberger, J. B.: 'JUnit Recipes' (Manning Publications Co., 2004, 1st edn.)

[13] Gaelli, M., Wamper, R., Nierstrasz, O.: 'Composing Tests from Examples', J Object Tech., 2007, 6, (9), pp. 71–86

[14] Huang, C.H., Huo, Y.C.: 'A semi-automatic generator for unit testing code files based on JUnit'. Proc. Int. Conf. on Systems, Man and Cybernetics, Waikoloa, USA, October 2005, pp. 140–145

[15] Janzen, D.S.: 'An Empirical Evaluation of the Impact of Test-Driven Development on Software Quality'. PhD thesis, University of Kansas, 2006

[16] Rompaey, B.V., Bois B.D., Demeyer, S.: 'Characterizing the Relative Significance of a Test Smell'. Proc. Int. Conf. on Software Maintenance, Philadelphia, USA, September 2006, pp. 391–400

[17] Loughran, S., Hatcher E.: 'Ant in Action' (Manning Publications Co., 2007, 2nd edn.)

[18] Offutt, J., Ma, Y.S., and Kwon, Y.R.: 'The Class-Level Mutants of MuJava'. Proc. Int. Workshop Automation of Software Test, Shanghai, China, May 2006, pp. 78–84

[19] Kim, S., Clark, J.A., and McDermid, J.A.: 'Class Mutation: Mutation Testing for Object-Oriented Programs'. Proc. Net.ObjectDays, Erfurt, Germany, October 2000, http://www-users.cs.york.ac.uk/~jac/papers/ClassMutation.pdf, accessed March 2008

[20] Chevalley, P.: 'Applying Mutation Analysis for Object-Oriented Programs Using a Reflective Approach'. Proc. 8th Asia-Pacific Software Engineering Conf., Washington, USA, December 2001, pp. 267–270

[21] King, K. N., Offutt A. J.: 'A Fortran Language System for Mutation-based Software Testing', Softw. Pract. Exper., 1991, 21, (7), pp. 685–718

[22] Offutt, A.J., Lee, A., Rothermel, G., Untch, R.H., and Zapf, C.: 'An Experimental Determination of Sufficient Mutant Operators', ACM Trans. Softw. Eng. and Meth., 1996, 5, (2), pp. 99–118

[23] Andrews, J.H., Briand, L.C., and Labiche, Y.: 'Is Mutation an Appropriate Tool for Testing Experiments?'. Proc. 27th Int. Conf. on Software Engineering, St Louis, USA, May 2005, pp. 402–411

[24] Andrews, J.H., Briand, L.C., Labiche, Y., Namin, A.S.: 'Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria'. IEEE Trans. Softw. Eng., 2006, 32, (8), pp. 608–624

[25] Walsh, P.J.: 'A Measure of Test Case Completeness'. PhD thesis, University New York, 1985

[26] Frankl, P.G., Weiss, S.N., Hu, C.: 'All-Uses vs Mutation Testing: An Experimental Comparison of Effectiveness'. J. Syst. Softw. 1997, 38, (3), pp. 235–253

[27] Offutt, A.J., Pan, J., Tewary, K., Zhang, T.: 'An Experimental Evaluation of Data Flow and Mutation Testing'. Softw. Pract. and Exper., 1996, 26, (2), pp. 165–176

[28] Offutt, A.J., Rothermel, G., and Zapf, C.: 'An Experimental Evaluation of Selective Mutation'. Proc. 15th Int. Conf. on Software Engineering, Baltimore, USA, May 1993, pp. 100–107

[29] Wong, W.E., Mathur, A.P.: 'Reducing the cost of mutation testing: An empirical study'. J. Syst. Softw. 1995, 31, (3), pp. 185–196

[30] DeMillo, R.A., Guindi, D.S., King, K.N., M.McCracken, W., Offutt, A.J.: 'An extended overview of the Mothra software testing environment'. Proc. 2nd Workshop on Software Testing, Verification, and Analysis, Banff, Canada, July 1988, pp. 142–151

[31] Delamaro, M.E., Maldonado, J.C.: 'Proteum – A Tool for the Assessment of Test Adequacy for C Programs'. Proc. Conf. on Performability in Computing Systems, East Brunswick, USA, July 1996, pp. 75–95

[32] Moore, I.: 'Jester - a JUnit test tester'. Proc. 2nd Int. Conf. on Extreme Programming and Flexible Processes in Software Engineering, Sardinia, Italy, May 2001, pp. 84–87

[33] Irvine, S. A., Tin, P., Trigg L., Cleary, J. G., Inglis, S., and Utting, M.: 'Jumble Java Byte Code to Measure the Effectiveness of Unit Tests'. Proc. Testing: Academic and Industrial Conf. Practice and Research Techniques, Windsor, UK, September 2007, pp. 169–175

[34] Offutt, J., Ma, Y.S., and Kwon, Y.R.: 'An Experimental Mutation System for Java', SIGSOFT Softw. Eng. Notes, 2004, 29, (5), pp. 1–4

[35] Ma, Y.S., Offutt, J., and Kwon, Y.R.: 'MuJava: A Mutation System for Java'. Proc. 28th Int. Conf. on Software Engineering, Shanghai, China, May 2006, pp. 827–830

[36] Chevalley, P., Thévenod-Fosse, P.: 'A mutation analysis tool for Java programs', Int. J. Softw. Tools Tech. Transfer, 2003, 5, (1), pp. 90–103

[37] Bogacki, B., and Walter, B.: 'Aspect-Oriented Response Injection: An Alternative to Classical Mutation Testing'. Proc. IFIP Work. Conf. on Software Engineering Techniques, Warsaw, Poland, October 2006, pp. 273–282

[38] Untch, R.H., Offutt A.J., and Harrold, M.J.: 'Mutation Analysis Using Mutant Schemata'. Proc. Int. Symp. on Software Testing and Analysis, Cambridge, USA, June 1993, pp. 139–148

[39] Ma, Y.S., Harrold, M.J., and Kwon, Y.R.: 'Evaluation of Mutation Testing for Object-Oriented Programs'. Proc. 28th Int. Conf. on Software Engineering, Shanghai, China, May 2006, pp. 869–872

[40] Ma, Y.S., Offutt, J.: 'Description of Class Mutation Mutation Operators for Java', November 2005, `http://cs.gmu.edu/~offutt/mujava/mutopsClass.pdf`

[41] Ma, Y.S., Offutt, J.: 'Description of Method-level Mutation Operators for Java', November 2005, http://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf

[42] Smith, B.H., and Williams, L.: 'An Empirical Evaluation of the MuJava Mutation Operators'. Proc. Testing: Academic and Industrial Conference, Windsor, UK, September 2007, pp. 193–202

[43] Venners, B.: 'Test-Driven Development. A Conversation with Martin Fowler, Part V'. http://www.artima.com/intv/testdrivenP.html, accessed September 2007

[44] Offutt, J.: 'An analysis of Jester based on published papers'. http://cs.gmu.edu/~offutt/jester-anal.html, accessed September 2006

[45] Ammann, P., and Offutt, J.: 'Introduction to Software Testing' (Cambridge University Press, 2008, 1st edn.)

[46] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., and Irwin, J.: 'Aspect-Oriented Programming'. Proc. European Conf. on Object-Oriented Programming, Jyväskylä, Finland, June 1997, pp. 220–242

[47] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., and Wesslén, A.: 'Experimentation in Software Engineering: An Introduction' (Kluwer Academic Publishers, 2000, 1st edn.)

[48] Rothfuss, G.J.: 'A Framework for Open Source'. MSc thesis, University of Zurich, 2002

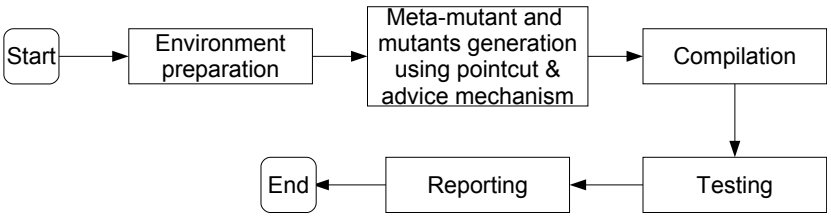

Figure 1: A schematic illustration of mutation testing using the FAMTA Light approach

```
1 public class ClassA {
2   public boolean isEven(int number){
3     if(number % 2 == 0){
4       return true;
5     } else {
6       return false;
7     }
8   }
9 }
```

Figure 2: Sample class named ClassA

```
1  public class ClassA {
2    public boolean isEven(int number){
3      if(number % 2 == 0){
4        return true;
5      } else {
6        return false;
7      }
8    }
9    public boolean isEven_mutation0(int number){
10     if(number % 2 != 0) {
11       return true;
12     } else {
13       return false;
14     }
15   }
16   public boolean isEven_mutation1(int number){
17     if(number % 2 == 0 && false ) {
18       return true;
19     } else {
20       return false;
21     }
22   }
23 }
```

Figure 3: A mutated `ClassA` class.

```
1 public privileged aspect ClassAAspect {
2   public static int counter = 0;
3
4   pointcut isEvenPointcut(ClassA thisObject, int number) :
5     execution(boolean somePackage.ClassA.isEven(int))
6       && !within(somePackage.ClassAAspect)
7       && target(thisObject)
8       && args(number);
9
10  boolean around(ClassA thisObject, int number)  :
11    isEvenPointcut(thisObject, number){
12    switch(ClassAAspect.counter){
13    case 0:
14      return thisObject.isEven_mutation0(number);
15    case 1:
16      return thisObject.isEven_mutation1(number);
17    default:
18      return thisObject.isEven(number);
19  }
20 }
```

Figure 4: A meta-mutant for the `ClassA` class.

```
1  for(every-tested-class) do
2    Initialize class test result;
3    //number of mutants = 0, number of killed mutants = 0
4    Get meta-mutant aspect for the tested class;
5    Get 'counter' and 'MAX_COUNTER' values from the aspect;
6    Get tests suite for the tested class;
7
8    for(every-class-mutant) do
9      Execute test suite;
10     if(any test in test suite failed or test suite
11       runs longer then maximum run time) then
12       Add killed mutant;
13       //number of mutants++, number of killed mutants++
14     else
15       Add alive mutant;
16       //number of mutants++
17     end if;
18     Add 1 to static meta-mutant 'counter' field value
19     //Static fields can be modified using reflection API
20   end for;
21
22   Set 'counter' field value to -1;
23   Print test results for class;
24 end for;
```

Figure 5: The implementation of the FAMTA Light mutants testing phase (pseudo-code)

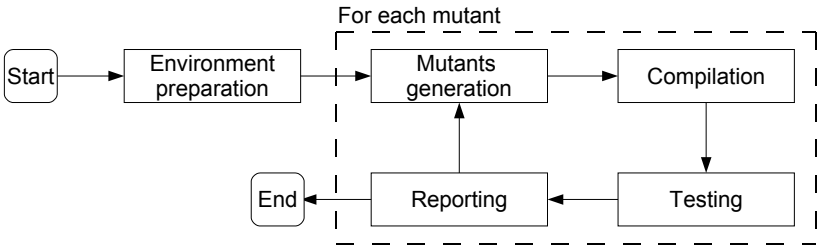

Figure 6: A schematic illustration of mutation testing using Jester

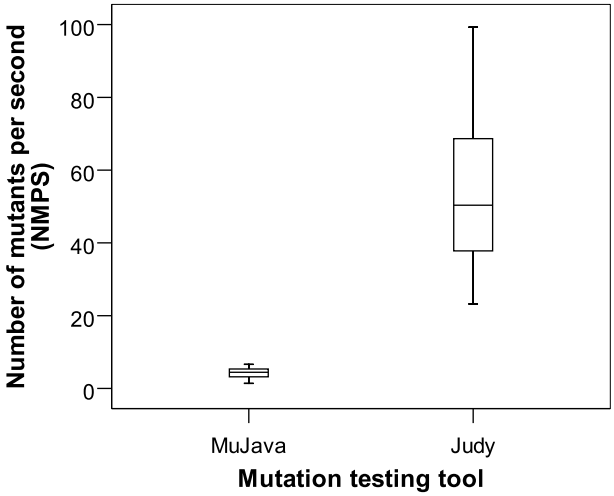

Figure 7: Number of mutants per second box plot