Mutation Testing on an Object–Oriented Framework: An Experience Report

Sergio Segura[*,a], Robert M. Hierons[b], David Benavides[**,a], Antonio Ruiz-Cortés[a]

[a] *Department of Computer Languages and Systems, University of Seville*
Av Reina Mercedes S/N, 41012 Seville, Spain
[b] *School of Information Systems, Computing and Mathematics, Brunel University*
Uxbridge, Middlesex, UB7 7NU United Kingdom

Abstract

Context: The increasing presence of Object–Oriented (OO) programs in industrial systems is progressively drawing the attention of mutation researchers toward this paradigm. However, while the number of research contributions in this topic is plentiful, the number of empirical results is still marginal and mostly provided by researchers rather than practitioners.
Objective: This article reports our experience using mutation testing to measure the effectiveness of an automated test data generator from a user perspective.
Method: In our study, we applied both traditional and class-level mutation operators to FaMa, an open source Java framework currently being used for research and commercial purposes. We also compared and contrasted our results with the data obtained from some motivating faults found in the literature and two real tools for the analysis of feature models, FaMa and SPLOT.
Results: Our results are summarized in a number of lessons learned supporting previous isolated results as well as new findings that hopefully will motivate further research in the field.
Conclusion: We conclude that mutation testing is an effective and affordable technique to measure the effectiveness of test mechanisms in OO systems. We found, however, several practical limitations in current tool support that should be addressed to facilitate the work of testers. We also missed specific techniques and tools to apply mutation testing at the system level.

Key words: Mutation testing, test adequacy, test data generation, automated analysis, feature models

1. Introduction

Mutation testing [10] is a fault-based testing technique that measures the effectiveness of test cases. First, faults are introduced in a program creating a collection of faulty versions, called *mutants*. The mutants are created from the original program by applying changes to its source code (e.g. i + 1 ⇒ i - 1). Each change is determined by a *mutation operator*. Test cases are then used to check whether the mutants and the original program produce different responses, detecting the fault. The number of mutants detected provides a measure of the quality of the test suite called *mutation score*.

Mutation testing has traditionally been applied to procedural programs written in languages like Fortran or C. However, the increasing presence of Object–Oriented (OO) programs in industrial systems is progressively drawing the attention of mutation researchers toward this paradigm [17]. Contributions in this context mainly focus on the development of new tools and mutation operators (class-level operators) specifically designed to create faults involving typical OO features like inheritance or polymorphism. However, little research has been done to study the effectiveness and limitations of OO mutation in practice.

[*] Principal corresponding author
[**] Corresponding author
Email addresses: sergiosegura@us.es (Sergio Segura), benavides@us.es (David Benavides)

The high cost of mutation has traditionally hindered its application in empirical studies, including those involving OO systems. To make it affordable, some researchers apply it to basic or teaching programs [1, 18, 36, 37] rather than real-world applications. Others, save effort by mutating only a part of the system [1, 19, 21, 26, 36, 37] or using a subset of mutation operators [12, 15, 18, 26, 33]. In both cases, the representativeness of the results is therefore only partial. Beside this, related studies are in most cases reported by mutation experts rather than practitioners, whose experiences could probably provide more insights about the applicability of the approach. Finally, although some interesting results have been provided, these are often isolated since they are extracted from a single system. This leads authors to concede the need for more practical experiences that support, contrast or complement their results [19, 21, 37].

In this article, we report our experience using mutation testing to measure the effectiveness of an automated test data generator for the analysis of feature models. Our work contributes to the field of practical experimentation with mutation as follows:

- We conducted the experiments using FaMa [5, 41], an open source Java OO framework for the analysis of feature models. FaMa is a widely-used tool of significant size under continuous development. It is already integrated into the feature modelling tool MOSKitt [8] and is being integrated into the commercial tool pure::variants[1] [34]. We are also aware of its use in different universities and laboratories for teaching and research purposes.
- We fully mutated three of the analysis components (so-called reasoners) integrated into FaMa using both traditional and class–level mutation operators for Java.
- We compared and contrasted our mutation results with the data from some motivating faults found in the literature and recent releases of FaMa and SPLOT [28], two real tools for the analysis of feature models. We are not aware of any other related work on mutation reporting results of real faults in OO programs.
- Equivalent mutants were detected and examined manually, rather than using partially-effective automated mechanisms [22, 33], providing helpful feedback about the detection effort and equivalence causes.
- To the best of our knowledge, this is the first work reporting the experience with mutation testing from a user perspective, leading to a number of lessons learned that could be helpful for both researchers and practitioners in the field.

The rest of this article is structured as follows. Section 2 provides an overview of the analysis of feature models and the FaMa Framework. The experimental setup used in our study and the analysis of results obtained are detailed in Sections 3 and 4 respectively. Section 5 compares and contrasts the mutation results with the data obtained when using our test data generator to detect some real faults in FaMa and SPLOT. In Section 6, we report our main findings in a number of lessons learned. The threats to validity of our study are presented in Section 7. Section 8 provides an overview of related studies. Finally, we summarize our main conclusions in Section 9.

2. FaMa framework

A *feature model* defines the valid combinations of features in a domain. These are widely used to represent the commonalities and variabilities within the products of a software product line [3]. The automated analysis of feature models is an active research area that deals with the computer–aided extraction of information from feature models. These analyses are generally performed in two steps. First, the model is translated into a specific logic representation (e.g. propositional formula). Then, off-the-shelf solvers are used to automatically perform a variety of *analysis operations* [4] on the logic representation of the model.

[1]This integration is being performed in the context of the DiVA European project (http://www.ict-diva.eu/)

FaMa [5, 41] is an open source Java framework for the automated analysis of feature models. FaMa provides a number of extension points to plug in new analysis components called reasoners. A *reasoner* is an extension of the framework implementing one or more analysis operations using a specific paradigm. Currently, FaMa integrates ready-to-use reasoners enabling the analysis of feature models using SAT solvers (SAT), Binary Decision Diagrams solvers (BDD) and Constraint Programming solvers (CP). A simplified version of the FaMa architecture is shown in Figure 1.

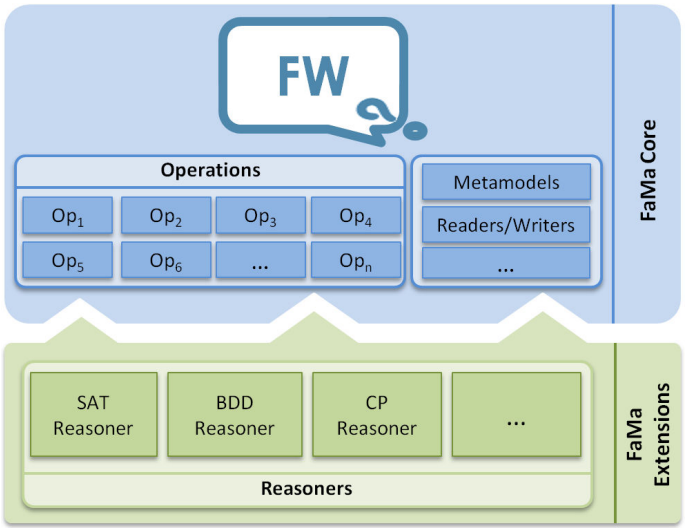


Figure 1: FaMa Architecture

FaMa is a widely-used tool of significant size under continuous development[2]. It is already integrated into the feature modelling tool MOSKitt [8] and is being integrated into the commercial tool pure::variants [34]. We are also aware of its use in different universities and laboratories for teaching and research purposes. These reasons, coupled to our familiarity with the tool as leaders of the project, made us choose FaMa as a good candidate to be mutated. More specifically, we used three of the reasoners integrated into FaMa as subject tools for our study.

3. Experimental setup

We next provide a full description of the mutation tool, mutation operators, subject programs, test data generator and experimental procedure used in the evaluation of our automated test data generator.

3.1. Mutation tool

For the generation of mutants, we used the MuClipse Eclipse plug-in v1.3 [37]. MuClipse is a visual Java tool for object-oriented mutation testing based on MuJava (Mutation System for Java) [21, 26]. We found this tool to provide helpful features for its use in our work, namely: *i*) wide range of mutation operators (including both traditional and class-level operators), *ii*) visual interface, especially useful for the examination of mutants and their statuses (i.e. alive or killed), *iii*) full integration with the development environment, and *iv*) support for the generation and execution of mutants in two clearly separated steps. Despite this, we found some limitations in the current version of the tool. Firstly, it does not support Java 1.5 code features. This forced us to make slight changes in the code, basically removing annotations and generic types when needed. Secondly, the tool did not support jar files. This was solved by manually placing the binary code of the application in the corresponding folder prior to the generation of mutants.

[2]Lines of code: 22,723. Developers: 6. Total releases: 8. Frequency of releases: 5 months.

Regarding the execution of mutants, we found that MuClipse and other related tools were not sufficiently flexible, providing as results mainly mutation score and a list of alive and killed mutants. To address our needs, we developed a custom execution module providing some extra functionality, namely: *i*) custom results such as time required to kill each mutant and number of mutants generated by each operator, *ii*) results in Comma Separated Values (CSV) format for its later processing in spreadsheets, and *iii*) fine-grained filtering capabilities to specify which mutants should be considered or ignored during the execution. For each class under evaluation, our execution module works in two iterative steps. First, the binary file of the original class is replaced by the corresponding file of the current mutant. Then, test cases are run and results collected using the programmatic API of JUnit 4 [38].

3.2. Mutation operators

There are two types of mutation operators for OO systems: *traditional* and *class-level* operators. The former are those adapted from procedural languages such as C or Fortran. These mainly mutate traditional programming features such as algebraic or logical operators (e.g. i $\Rightarrow$ i++). The latter are specifically designed to introduce faults affecting OO features like inheritance or polymorphism (e.g. *super* keyword deletion). In our study, we applied all 15 traditional and 28 class-level operators for Java available in MuClipse, listed in Appendix A. For details about these operators we refer the reader to [24, 25].

3.3. Experimental data

We chose three of the reasoners integrated into the FaMa Framework as subject tools for our experiments, namely: the Sat4jReasoner v0.9.2 (using satisfiability problems by means of Sat4j solver [6]), the JavaBDDReasoner v0.9.2 (using binary decision diagrams by means of JavaBDD solver [42]) and the JaCoPReasoner v0.8.3 (using constraint programming by means of JaCoP solver [20]). Each of these reasoners uses a different paradigm to perform the analyses and was coded by different developers, providing the required heterogeneity for the evaluation of our approach.

For each reasoner, we selected the classes extending the framework and implementing the analyses capabilities. Table 1 shows the number of classes selected from each reasoner together with the total number of lines of code[3] (LoC) and the average number of methods and attributes per class. The size of the 27 subject classes included in our study ranged between 35 and 220 LoC. These metrics were computed automatically using the plug-in Metrics 1.3.6 for Eclipse [39].

Reasoner	**LoC**	**Classes**	**Av Methods**	**Av Attributes**
Sat4jReasoner	743	9	6.5	1.8
JavaBDDReasoner	625	9	6.3	2.1
JaCoPReasoner	686	9	6.3	2.3
Total	**2,054**	**27**	**6.4**	**2.1**

Table 1: Size statistics of the three subject reasoners

For each reasoner, the implementations of six different analysis operations were tested. A description of these operations is given in Table 2.

3.4. Test data generator

Test cases were automatically generated using the automated test data generator presented by the authors in [35]. This is designed to detect faults in the implementations of the analysis operations on feature models regardless of how these are implemented. Thus, it uses a black-box testing approach relying on the inputs and outputs of the analysis operations under test. Given an initial test case, the tool automatically generates random follow-up test cases using known relations between the input feature models and their expected outputs (so-called metamorphic relations). The test generation process was parameterised by a number of

[3]LoC is any line within the Java code that is not blank or a comment.

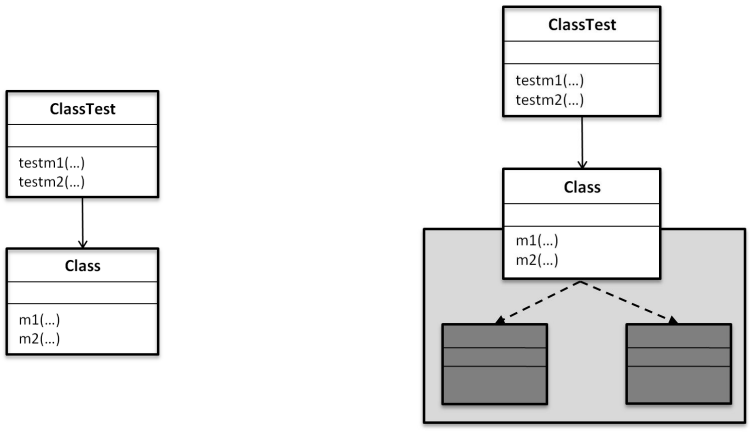
Operation	Description
VoidFM	Informs whether the input feature model is void or not (i.e. it represent no products)
ValidProduct	Informs whether the input product belongs to the set of products of a given model
Products	Returns the set of products of a feature model
#Products	Returns the number of products represented by a feature model
Variability	Returns the variability degree of a feature model
Commonality	Returns the percentage of products represented by the model in which a given feature appears

Table 2: Analysis operations tested

factors such as the number of features or the percentage of constraints of the input models to be generated. For each operation under test, we used our generator to generate and run test cases until a fault was found or a timeout was exceeded.

3.5. Mutants execution

Traditionally, classes are considered the units of functionality during mutation testing in OO systems (Figure 2(a)). That is, each class is first mutated and then their mutants are executed against a test set. In our study, however, the nature of FaMa and our generator forced us to revise the notion of unit testing. The goal of our test data generator is to detect faults in the implementation of analysis operations regardless of how these are implemented. Thus, we consider a testing unit as a black-box component providing the functionality of an analysis operation (Figure 2(b)). No knowledge is assumed about the internal structure of the component, only about the interface of the operation that it implements. Note that this means that a testing unit could therefore be composed of more than one class.



(a) Class-level testing (b) Component-level testing

Figure 2: Class-level vs. component-level testing

As an example, consider the partial class diagram of Sat4jReasoner showed in Figure 3. As previously explained, we can only assume knowledge about the classes extending the public interfaces of the operations provided by the framework i.e. these are the only classes we generated test data for. Note that the functionality of the operations is not provided by a single class, but several of them that are often reused by different operations. For instance, class Reasoner participates in the implementation of all the operations under test. Once all classes in the reasoner were mutated, we evaluated the ability of our generator to kill the mutants in a three-step process as follows:

1. We grouped those classes providing the functionality of each analysis operations, i.e. this could be considered a testing unit, a component. For instance, in Figure 3, classes {Variability + Reasoner + #Products} provide the functionality of the operation *Variability*.

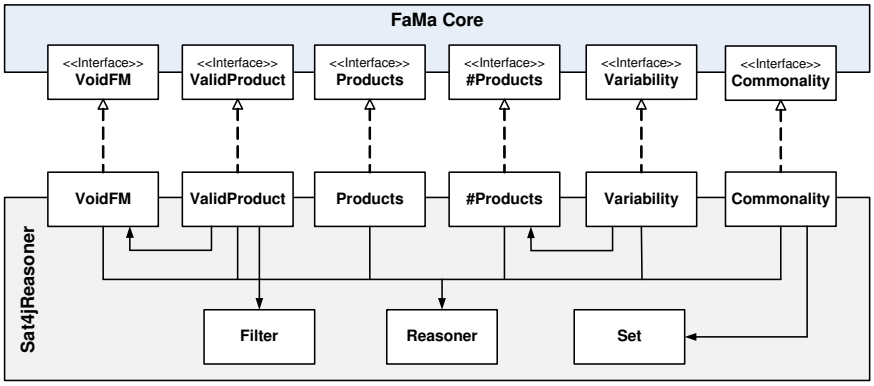


Figure 3: Partial class diagram of Sat4jReasoner (some associations are omitted for simplicity)

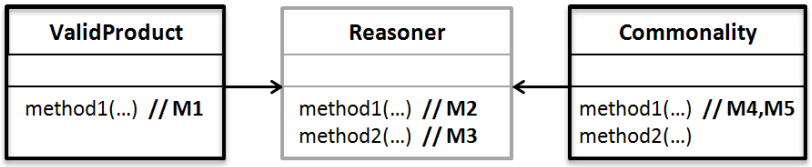
2. For each operation, we performed a set of executions trying to kill the mutants of the classes implementing the operation. On each execution, we mutated a single class and left the rest in the original form. For instance, let us use C_T to denote the set of mutants of a given class C. Tests in the operation *Variability* were evaluated by trying to kill mutants when testing the operation with the following combination of classes $\{\text{Variability}_T + \text{Reasoner} + \#\text{Products}\}$, $\{\text{Variability} + \text{Reasoner}_T + \#\text{Products}\}$ and $\{\text{Variability} + \text{Reasoner} + \#\text{Products}_T\}$.
3. To work out the results of each operation, we considered the results obtained on each of the executions associated to it.

The previous process raised new challenges to be solved during the execution and computation of results. To illustrate this, consider the simplified class diagram depicted in Figure 4. The Reasoner class is reused in the implementation of the operations *ValidProduct* and *Commonality*, i.e. it is shared by two different testing units. A label is inserted in those methods where a change was introduced generating a mutant, M < i >. We felt the need to distinguish clearly between the following terms:

- *Generated mutants.* These are the mutants generated in the source code, a total of 5 in Figure 4, i.e. ValidProduct_{M1}, Reasoner_{M2}, Reasoner_{M3}, Commonality_{M4} and Commonality_{M5}.
- *Executed mutants.* These are the mutants actually executed when considering the mutants in reusable classes. In the example, mutants in the reused class Reasoner (i.e. Reasoner_{M2} and Reasoner_{M3}) are executed against the test data of the operations *ValidProduct* and *Commonality* resulting in 7 mutants executed listed in the table of Figure 4.

In addition, we identified two new mutant statuses to be considered when processing our results, namely:

- *Uncovered mutants.* We say a mutant in a reusable class is uncovered by a given operation if it cannot be exercised by that operation. For instance, let us suppose that class ValidProduct in Figure 4 does not use *method2* of the class *Reasoner*. We would then say that mutant Reasoner_{M3} is not covered by the operation *ValidProduct* (see executed mutant 3 in Figure 4). Uncovered mutants are a type of equivalent mutants and were manually identified and omitted for the computation of results on each operation.
- *Partially equivalent mutants.* We say a mutant in a reusable class is partially equivalent if it is equivalent for a subset of the operations using it (but not all of them). For instance, Reasoner_{M2} is alive when tested against the tests of the operation *ValidProduct* (i.e. $\{\text{ValidProduct} + \text{Reasoner}_{M2}\}$) but equivalent when tested with the test data of the operation *Commonality* (i.e. $\{\text{Commonality} + \text{Reasoner}_{M2}\}$). In contrast to conventional equivalent mutants, partially equivalent mutants were not excluded from our study. Instead, they were included and omitted only in those operations where they were identified as equivalent.



Executed mutants	Status
1. ValidProduct$_{M1}$ + Reasoner	Killed
2. ValidProduct + Reasoner$_{M2}$	Alive
3. ValidProduct + Reasoner$_{M3}$	Uncovered
4. Commonality$_{M4}$ + Reasoner	Alive
5. Commonality$_{M5}$ + Reasoner	Killed
6. Commonality + Reasoner$_{M2}$	Equivalent
7. Commonality + Reasoner$_{M3}$	Killed

Figure 4: Example of mutation in reusable classes

3.6. Experimental procedure

For the application of mutation testing in FaMa, we followed four steps, namely:

1. *Reasoners testing.* Prior to their analysis, we checked whether the original reasoner passed all the tests. A timeout of 600 seconds was used. As a result, we detected and fixed a defect in the JaCoPReasoner. We found this fault to be especially motivating since it was also present in the studied release of FaMa (see Section 5.2 for details).
2. *Mutants generation.* We generated two sets of mutants by applying all traditional and class–level operators available in MuClipse (listed in Appendix A).
3. *Mutants execution.* For each executed mutant, we ran our test data generator until finding a test case that kills it or until a timeout of 600 seconds was exceeded. We chose this value for the timeout because it had proved to be effective in similar experiments with mutants and automated metamorphic testing [14].
4. *Results processing.* We classified mutants into different categories. Generated mutants were classified into equivalent, not equivalent, undecided and discarded mutants. We marked as *undecided* those mutants whose equivalence we could not conclude in a reasonable time (more details in Section 4.3). *Discarded* mutants were those changing parts of the programs exercised by the tests (they were not uncovered) but that were of no interest for our study since they affected secondary functionalities not addressed by our test data generator (e.g. execution time measurement). Executed mutants were classified into killed, alive, uncovered and partially equivalent mutants. Finally, test data generation and execution results were processed for each operation.

All our experiments were performed on a laptop machine equipped with an Intel Pentium Dual CPU T2370@1.73GHz and 2048 MB of RAM memory running Windows Vista Business Edition and Java 1.6.0_05.

4. Analysis of results

In this section, we report the analysis of our results. We first outline the data obtained with both traditional and class-level mutation operators. Then, we describe our experience detecting equivalent mutants.

4.1. Analysis of results with traditional mutants

Table 3 depicts information about the mutants obtained when applying traditional mutation operators. We distinguish between "Generated Mutants" and "Executed Mutants", where a generated mutant is a mutated class and an executed mutant is a component (i.e. set of classes implementing an operation) that

contains one mutated class (see Section 3.5 for details). In procedural programs, the number of generated mutants is typically large. For example, a simple Fortran program of 29 LoC computing the days between two dates results in 3,010 mutants [31]. In FaMa, however, the total number of generated mutants for the 27 subject classes was 749. This occurs because traditional mutants mainly modified features that the FaMa classes barely contained (e.g. arithmetic operators). Out of the total 749 mutants, 101 (13.4%) were identified as semantically equivalent. This percentage is within the boundaries reported in similar studies with procedural programs which suggest that between 5% and 15% of generated mutants are equivalent [11, 13, 30, 31, 32, 33]. In addition to these, we also discarded 87 mutants (11.6%) affecting other aspects of the program not related to the analysis of feature models and therefore not addressed by our test data generator. These were mainly related to the computation of statistics (e.g. execution time) and exception handling.

Executed mutants only include killable mutants, i.e. not equivalent. The number of executed mutants was nearly twice the number of generated mutants. That means that many of the generated mutants were in reusable classes.

Operator	**Generated Mutants**			**Executed Mutants**			**Score (%)**
	Total	**Equivalent**	**Discarded**	**Total**	**Alive**	**Killed**	
AOIS	296	66	32	547	11	536	97.9
ROR	106	9	3	230	0	230	100
LOI	93	5	13	180	0	180	100
COI	87	5	3	180	0	180	100
AORB	68	7	24	58	0	58	100
AOIU	53	5	7	84	0	84	100
COD	19	0	5	21	0	21	100
AORS	18	0	0	57	0	57	100
COR	8	4	0	7	0	7	100
AODU	1	0	0	1	0	1	100
Total	**749**	**101**	**87**	**1,365**	**11**	**1,354**	**99.1**

Table 3: Traditional mutants for FaMa classes

Operators AODS, SOR, LOR, LOD and ASRS did not generate any mutants since they mutate language operators that the subject programs did not employ (e.g. unary logic). The operator AOIS produced the most mutants (39.5% of the total) followed by ROR (14.15%), LOI (12.4%) and COI(11.6%). The dominance of these four operators was also observed in related studies with Java programs [36, 37]. Regarding individual mutation scores, the operator AOIS, with a score of 97.9%, was the only one introducing faults not detected by our generator. Hence, this was the only operator providing helpful information to improve the quality of our tool.

Tables 4, 5 and 6 show the results obtained when using our test data generator to try to kill traditional mutants in the three subject reasoners. For each operation, the number of classes involved, number of executed mutants, test data generation results and mutation score are presented. Test data generation results include average and maximum time required to kill each mutant and average and maximum number of test cases generated before killing a mutant. These results provide us with quantitative data to measure and compare the effectiveness of our generator when killing mutants. Besides this, the average time and number of test cases generated before killing a mutant gives an idea of how difficult it was to detect faults in a certain operation. For instance, operations *Products*, *#Products*, *Variability* and *Commonality* showed a mutation score of 100% in all the reasoners with an average number of test cases required to kill each mutant under 2. This suggests that faults in these operations are easily killable. On the other hand, faults in the operations *VoidFM* and *ValidProduct* appeared to be more difficult to detect. We found that mutants on these operations required input models to have a very specific pattern in order to be revealed. As a consequence of this, the average time and number of test cases in these operations were noticeable higher than in the rest of analyses tested.

The maximum average time to kill a mutant was 7.4 seconds. In the worst case, our test data generator

spent 566.5 seconds before finding a test case that killed the mutant (operation *VoidFM* in Table 4). In this time, 414 different test cases were generated and run. This gave us an idea of the minimum timeout that should be used when applying our approach in real scenarios.

Operations		Executed Mutants		Test Data Generation				Score (%)
Name	**Classes**	**Total**	**Alive**	**Av Time (s)**	**Max time (s)**	**Av TCs**	**Max TCs**	
VoidFM	2	55	0	37.6	566.5	95.1	414	100
ValidProduct	5	109	3	4.3	88.6	12	305	97.2
Products	2	86	0	0.6	3.4	1.5	12	100
#Products	2	57	0	0.7	2.4	1.8	8	100
Variability	3	82	0	0.6	1.7	1.3	5	100
Commonality	5	109	0	0.6	3.8	1.5	13	100
Total	**19**	**498**	**3**	**7.4**	**566.5**	**18.9**	**414**	**99.3**

Table 4: Test data generation results using traditional operators in Sat4jReasoner

Operations		Executed Mutants		Test Data Generation				Score (%)
Name	**Classes**	**Total**	**Alive**	**Av Time (s)**	**Max time (s)**	**Av TCs**	**Max TCs**	
VoidFM	2	75	3	6.6	111.7	29.3	350	96
ValidProduct	5	129	5	1	34.6	3.8	207	96.1
Products	2	130	0	0.7	34.6	1.4	12	100
#Products	2	77	0	0.5	1.4	1.6	6	100
Variability	3	104	0	0.5	2.4	1.6	12	100
Commonality	5	131	0	0,5	3	1.5	16	100
Total	**19**	**646**	**8**	**1.6**	**111.7**	**6.5**	**350**	**98.7**

Table 5: Test data generation results using traditional operators in JavaBDDReasoner

Operations		Executed Mutants		Test Data Generation				Score (%)
Name	**Classes**	**Total**	**Alive**	**Av Time (s)**	**Max time (s)**	**Av TCs**	**Max TCs**	
VoidFM	2	8	0	1.5	8.3	11.3	83	100
ValidProduct	5	61	0	0.7	1.2	1.3	5	100
Products	2	37	0	0.5	0.7	1	1	100
#Products	2	13	0	0.5	0.7	1	1	100
Variability	3	36	0	0.5	0.7	1	1	100
Commonality	5	66	0	0.5	0.7	1.1	3	100
Total	**19**	**221**	**0**	**0.7**	**8.3**	**2.8**	**83**	**100**

Table 6: Test data generation results using traditional operators in JaCoPReasoner

4.2. Analysis of results with class mutants

Table 7 shows information about the mutants obtained from class–level mutation operators. The total number of mutants is nearly 60% lower than the number of traditional mutants. This suggest that the OO features that these operators modify are less frequent than those features addressed by traditional operators such as arithmetic or relational operators. Out of the total 310 mutants, 141 (45.4%) were identified as semantically equivalent, far more than the 5% to 15% found in related studies with procedural programs. In certain cases, we could not conclude whether a mutant was equivalent or not. We marked these mutants, a total of 10 (3.2%), as "undecided". Finally, we discarded 10 (3.2%) mutants affecting secondary functionality of the programs not related to the analyses and not addressed by our test data generator.

Class–level operators IHD, IOR, ISI, ISD, IPC, PMD, PPD, PCC, OMR, OMD, OAN, JSD, JID, EOA and EOC did not generate any mutants and therefore they are not showed in the table. We found that these operators mainly mutate inheritance and polymorphism features that the subject classes did not use. The

Operator	Generated Mutants				Executed Mutants			Score (%)
	Total	Equivalent	Undecided	Discarded	Total	Alive	Killed	
PCI	118	93	0	0	108	0	108	100
IOD	55	13	0	0	60	0	60	100
JSI	47	14	0	0	52	52	0	0
EAM	20	2	0	6	44	0	44	100
JDC	19	11	0	0	14	0	14	100
PNC	12	2	6	0	12	0	12	100
EMM	10	0	4	0	12	12	0	0
PRV	9	0	0	2	26	0	26	100
JTI	6	0	0	1	5	0	5	100
PCD	5	5	0	0	0	0	0	N/A
JTD	4	0	0	1	3	0	3	100
IHI	3	1	0	0	3	0	3	100
IOP	2	0	0	0	3	0	3	100
Total	**310**	**141**	**10**	**10**	**342**	**64**	**278**	**81.2**

Table 7: Class mutants for FaMa classes

operator PCI produced the most mutants with nearly 40% of the total. This operator inserts type casting to variables (e.g. feature ⇒ (GenericFeature) feature) and therefore is likely to be applied more frequently than other operators. Operators PCI and PNC generated the highest percentages of equivalent mutants with 78.8% and 57.8% respectively.

Operators PCI, IOD, EAM, JDC, PNC, PRV, JTI, JTD, IHI and IOP got a mutation score of 100%. This suggests that the mutants generated by these operators in FaMa can be easily killed with the same test cases generated for traditional mutants. These operators were therefore useless in terms of providing information to improve the quality of our tests. On the other hand, operators JSI and EMM got a mutation score of 0% providing us with information to improve our tool. For instance, we learned that using sequences of calls to several instances of the same class (instead of one instance per test case) would allow us to kill those mutants generated by the operator JSI (*static* modifier insertion), and related faults, making our test suite stronger.

Tables 8, 9 and 10 show the results obtained when using our test data generator to try to kill the class mutants. The mutation scores on each solver ranged from 69.6% to 91.9%. This means that our generator, despite not being specifically designed to detect OO faults, was able to detect a majority of the class mutants introduced in FaMa. The average time to kill each mutant was between 0.7 and 1.5 seconds. Similarly, the average number of test cases generated before killing each mutant was between 4.4 and 4.7. These results are better than the ones obtained when using our generator with traditional mutants. This reveals that class mutants that were killed, were also easier to kill than traditional ones.

Operations		Executed Mutants		Test Data Generation				Score (%)
Name	Classes	Total	Alive	Av Time (s)	Max time (s)	Av TCs	Max TCs	
VoidFM	2	10	3	5,7	37	21	141	70
ValidProduct	5	31	9	1.2	5.7	3	20	71
Products	2	12	3	0.7	0.9	1	1	75
#Products	2	10	3	0.6	0.7	1	1	70
Variability	3	12	4	0.6	0.9	1.2	2	66.7
Commonality	5	27	9	0.5	0.7	1	1	66.7
Total	**19**	**102**	**31**	**1,5**	**37**	**4.7**	**141**	**69.6**

Table 8: Test data generation results using class-level operators in Sat4jReasoner

4.3. Equivalent mutants

The detection of equivalent mutants was performed by hand investing no more than 15 minutes per mutant in the worst case. If this time was exceeded without reaching a conclusion the mutants was marked as

Operations		Executed Mutants		Test Data Generation				Score (%)
Name	**Classes**	**Total**	**Alive**	**Av Time (s)**	**Max time (s)**	**Av TCs**	**Max TCs**	
VoidFM	2	7	1	1.4	5.1	19.8	112	85.7
ValidProduct	5	26	7	0.5	1.1	1.4	4	73.1
Products	2	7	1	0.6	1	1.7	3	85.7
#Products	2	7	1	0.7	1	2.5	4	85.7
Variability	3	9	2	0.5	1	1.6	4	77.8
Commonality	5	24	8	0.5	0.9	1.4	3	66.7
Total	**19**	**80**	**20**	**0.7**	**5.1**	**4.7**	**112**	**75**

Table 9: Test data generation results using class-level operators in JavaBDDReasoner

Operations		Executed Mutants		Test Data Generation				Score (%)
Name	**Classes**	**Total**	**Alive**	**Av Time (s)**	**Max time (s)**	**Av TCs**	**Max TCs**	
VoidFM	2	21	0	3	31.2	20.5	226	100
ValidProduct	5	35	4	0.6	1.5	1.5	9	88.6
Products	2	22	1	0.6	1.3	1	1	95.5
#Products	2	21	1	0.6	1	1.3	5	95.2
Variability	3	24	2	0.7	1.1	1.3	5	91.7
Commonality	5	37	5	0.6	1.4	1.1	4	86.5
Total	**19**	**160**	**13**	**1**	**31.2**	**4.4**	**226**	**91.9**

Table 10: Test data generation results using class-level operators in JaCoPReasoner

"undecided". In many cases, equivalent mutants were caused by similar structures in different classes. Once an equivalent mutant was detected, we found that detection of similar equivalent mutants was easier and faster. This suggests that the effort required to identify equivalent mutants is affected by the commonalities of the subject classes.

Regarding hardness, we found it especially difficult to determine the equivalence in those mutants replacing methods calls. In our study, these were generated by the class–level operators EMM and PNC. As an example, consider the following mutant generated by the operator PNC in Sat4jReasoner: new DimacsReader(solver) ⇒ new CardDimacsReader(solver). The mutant changes the default reader used by Sat4j (off-the-shelf solver used by Sat4jReasoner) to process input files. Determining whether the functionality of the new reader is equivalent to the original one for any input problem was not trivial, even when checking the available documentation of Sat4j. This and other related mutants were therefore marked as "undecided".

5. Real faults

For a further evaluation of our approach, we checked the effectiveness of our tool in detecting real faults. This allowed us to study the representativeness of mutants when compared to faults identified in real scenarios. In particular, we first studied a motivating fault found in the literature. Then, we used our generator to test recent releases of two tools, FaMa and SPLOT, detecting two defects in each of them. These results are next reported.

5.1. A motivating fault found in the literature

Consider the work of Batory in SPLC'05 [3], one of the seminal papers in the community of automated analysis of feature models. The paper included a bug (later fixed[4]) in the mapping of feature models to propositional formulas. Although this is not a real fault (i.e. it was not found in a tool), it represents a real type of fault likely to appear in real scenarios. This fault is motivational for two main reasons, namely: *i*) it affects all the analysis operations using the wrong mapping and *ii*) it is difficult to detect since it can

[4] ftp://ftp.cs.utexas.edu/pub/predator/splc05.pdf

only be revealed with input feature models containing a very specific pattern. For more details about this fault we refer the reader to [35]. We implemented this wrong mapping into a mock reasoner for FaMa using JavaBDD and tried to detect the fault using our test data generator.

Table 11 depicts the results of the evaluation. The testing procedure was similar to the one used with mutation testing. A maximum timeout of 600 seconds was used. All the results are the average of 10 executions. The fault was detected in all operations remaining latent in 50% of the tests performed in the *ValidProduct* operation. When examining the data, we concluded that this was due to the basic strategies used in our test data generator for the selection of inputs products for this operation. We presume that using more complex heuristics for this purpose would improve the results.

Operation	**Av Time (s)**	**Max Time (s)**	**Av TCs**	**Max TCs**	**Score (%)**
VoidFM	78.2	229.1	515.8	905	100
ValidProduct	38.4	43.7	268.4	322	50
Products	1.1	2.9	5.7	19	100
#Products	1.0	2.7	5.4	16	100
Variability	1.2	2.1	6.4	13	100
Commonality	1.4	3	7.8	20	100
Total	**20.2**	**229.1**	**134.9**	**905**	**91.6**

Table 11: Test data generation results using a motivating fault reported in the literature (averages of 10 executions)

The average time to detect the fault (20.2 s) was far higher than the average times obtained when killing mutants, ranging between 0.7 and 7.4 seconds. Similarly, the average number of test cases generated before detecting the fault (134.9) exceeded the averages obtained in the execution of mutants ranging between 4.4 and 18.9. Thus, our results reveal that detecting the fault in our mock tool was harder than killing the FaMa mutants.

5.2. FaMa Framework

We also evaluated our tool by trying to detect faults in a recent release of the FaMa Framework, *FaMa v1.0 alpha*. A timeout of 600 seconds was used for all the operations since we did not know *a priori* the existence of faults. Tests revealed two defects. The first one, also detected during our experimental work with mutation, was caused by an unexpected behaviour of JaCoP solver when dealing with certain heuristics and void models in the operation *Products*. In these cases, the solver did not instantiate an array of variables raising a null pointer exception. The second fault affected the operations *ValidProduct* and *Commonality* in Sat4jReasoner. The source of the problem was a bug in the creation of propositional clauses in the so-called staged configurations, a new feature of the tool.

Table 12 shows the results of our tests in FaMa. The results are the average of the data obtained in 10 executions. As illustrated, the defect in Sat4jReasoner was easy to detect in almost all cases with just one test case. On the other hand, the fault in JaCoPReasoner was harder to detect with a detection time of 160.9 seconds and 391.1 test cases generated on average. These results are again much higher than the averages obtained with mutation suggesting that this fault was harder to detect than mutants.

Operation	**Av Time (s)**	**Max Time (s)**	**Av TCs**	**Max TCs**	**Score (%)**
JaCoPReasoner - Products	160.9	214.9	391.1	535	100
Sat4jReasoner - ValidProduct	0.9	1.2	1	1	100
Sat4jReasoner - Commonality	0.9	1.3	1.1	2	100
Total	**54.2**	**214.9**	**131.1**	**535**	**100**

Table 12: Test data generation results in FaMa 1.0 alpha (averages of 10 executions)

5.3. SPLOT

Software Product Lines On-line Tools (SPLOT) [28] is a Web portal providing a complete set of tools for on-line editing, analysis and storage of feature models. It supports a number of analyses on cardinality-based feature models using propositional logic by means of the Sat4j and JavaBDD solvers. The authors of SPLOT kindly sent us a standalone version[5] of their system to evaluate our automated test data generator. In particular, we tested the operations *VoidFM*, *#Products* and *DeadFeatures* in SPLOT. As with FaMa, we used a timeout of 600 seconds and tested each operation 10 times to get averages. Tests revealed two defects in all the executions. The first one affected all operations on the SAT-based reasoner. With certain void models, the reasoner raised an exception (*org.sat4j.specs.ContradictionException*) and no result was returned. The second bug was related with cardinalities in the BDD-based tool. We found that the reasoner was not able to process cardinalities other than [1,1] and [1,*]. As a consequence of this, input models including or-relationships specified as [1,n] (n being the number of subfeatures) caused a failure in all the operations tested.

Table 13 depicts the results of our tests in SPLOT. Notice that the *DeadFeatures* operation was only implemented in the SAT-based reasoner. Detection times were even lower than those found with traditional mutants in FaMa suggesting that the bugs were easier to detect than mutants. We may remark, however, that SPLOT was much faster than FaMa in executing test cases and therefore time does not provide a fair comparison. From the average number of test cases, however, we found that the fault in the SAT-based reasoner required the generation of more test cases (between 26.7 and 38.3 on average) than FaMa mutants suggesting that this fault was slightly harder to detect than mutants in general. The fault in the BDD-based reasoner, on the other hand, was trivially detected in all cases.

Operation	**Av Time (s)**	**Max Time (s)**	**Av TCs**	**Max TCs**	**Score (%)**
Sat4jReasoner - VoidFM	0.7	1.3	26.7	66	100
Sat4jReasoner - #Products	1	2	26.1	66	100
Sat4jReasoner - DeadFeatures	0.9	2.2	38.3	134	100
JavaBDDReasoner - VoidFM	0.4	0.5	1.5	2	100
JavaBDDReasoner - #Products	0.4	0.5	1.9	5	100
Total	**0.7**	**2.2**	**18.9**	**134**	**100**

Table 13: Test data generation results in SPLOT (averages of 10 executions)

Faults detected in the standalone version of the tool were also observed in the online version of SPLOT. We may remark that authors confirmed the results and told us they were aware of these limitations.

6. Discussion and lessons learned

Our results can be summarized in the following lessons learned:

The number of mutants was lower than in procedural programs. The number of mutants generated by traditional mutation operators in FaMa was much lower than the number of mutants generated in smaller procedural programs [11, 13, 30, 31, 32]. This was also observed in related studies with OO systems [19, 21, 36, 37]. Like FaMa, the OO systems used in related studies do not perform many arithmetic or logical operations keeping the number of traditional mutants lower than in procedural programs. This suggests that mutation testing could be affordable when applied to OO applications with a reduced number of these features. This suggests that it would be useful to have a prediction model (e.g. equation) that practitioners could use to estimate the cost of mutation according to the characteristics of their programs. While some progress has been made in predicting the number of mutants in procedural languages [31], this seems to remain a challenge in the context of OO programs in which only some preliminary analysis has

[5]SPLOT does not use a version naming system. We tested the tool as it was in February 2010.

been proposed. [23].

The number of class mutants was lower than the number of traditional mutants. The number of mutants generated by class–level mutation operators was lower (about 60%) than traditional mutants. This trend was also observed in related studies with OO systems [12, 19, 21, 22, 26, 33]. This suggests that the OO features mutated by class-level operators occur less frequently than those addressed by traditional ones. It is worth remarking that even if we consider both traditional and class mutants in our study, there were still far fewer mutants than have been reported with smaller procedural programs. This also supports the applicability of mutation testing in OO systems.

Class–level operators generated more equivalent mutants than traditional ones. Our results show that the percentage of equivalent mutants generated by class–level operators (i.e. 45.4%) is much higher than the one generated with traditional operators (i.e. 13.4%). This result is also higher than the percentages reported in similar studies with procedural programs suggesting that between 5% and 15% of generated mutants are equivalent. This makes class-level mutation operators less attractive for experimentation since they generate fewer mutants than traditional operators and a larger percentage of equivalent ones reducing the total portion of useful (i.e. killable) mutants. Having said this, we may remark that we found contradictory results in the literature. On the one hand, studies conducted by Kim *et al.* [19] and Ma *et al.* [21] reported percentages of equivalent mutants of 4.9% (23 out of 466) and 0.4% (3 out of 691) respectively, much less than the percentage found in our study. On the other hand, a more recent experiment conducted by Ma *et al.* [22] revealed that at least 86% of the class mutants were equivalent, far more than the percentage found in our evaluation. We may mention that their experiment was performed on six open source programs with a total of 49,071 mutants studied which makes their results stronger than previous findings. When studying this work, we found that 9 of the mutation operators that generated higher percentages of equivalent mutants in their study (73.1% on average) did not generate any mutants in our work. We think this explains why the percentage of equivalent mutants in our work is lower supporting their result. Nevertheless, the heterogeneous data found in the literature suggest that a more extensive experiment using a wider number of subject programs would be highly desirable to shed light on this issue.

A majority of the class mutants were killed with the same kind of test cases designed for traditional mutants. Over 80% of the class mutants were killed with the same type of test data generated to kill traditional mutants. This was also observed by Ma et al. [21] in the BCEL system who found that more than 50% of the class mutants were killed with the same test cases used to kill traditional mutants. A further result in our study is that class mutants that were killed, were also easier to kill than traditional ones in terms of time invested by our generator to detect them. These results reveal that most class–level operators generated easily killable faults and therefore were not helpful in providing information to improve the quality of our test data generator. Only a small subset of these, JSI and EMM in our study, showed to be effective in providing feedback for the refinement of our tests.

Real faults were harder to detect than artificial ones. Three out of the five real faults studied in our work had average detection times or average number of test cases generated significantly higher than those of traditional and class mutants. Operation *ValidProduct*, for instance, got the lowest score (50%) when studying the motivating fault found in the literature (see Section 5.1). This suggests that real faults (based in our study) were, on average, harder to detect than mutants. We may remark, however, that this was not the case for all mutants. Hence, the highest detection time (566.5 seconds) was obtained when evaluation a mutant in the operation *VoidFM* of Sat4jReasoner (see Table 4). This means that mutants were able to represent faults as hard to detect as real ones. More importantly, the mutation scores obtained when studying real faults were similar (100% in the case of tests in FaMa 1.0 alpha and SPLOT) to those obtained with mutation in FaMa reasoners. This supports the *"competent programmer assumption"* that underlies the theory of mutation testing. That is, the assumption that faults made by programmers will be detected by those test suite able to kill mutants.

Relative cost of detecting equivalence mutants. The detection of equivalent mutants was easier and faster as we progressed and we found equivalent mutants similar to those previously identified. This means that the effort required to detect equivalent mutants was influenced by the similarities found in the subject programs. In our study, commonalities were frequent since the three reasoners implemented common interfaces provided by the framework. It would be interesting to know to what extent this happens in other tools and what impact this has on the cost of mutation testing.

Mutation testing on non–trivial testing units. Mutation testing in OO systems is traditionally applied at the class level, i.e. each single class is considered a unit of functionality. In our study, however, we consider a testing unit as a black–box component providing the functionality of an analysis operation. No knowledge is assumed about the internal structure of the component, only about the interface that it implements. Although possible, we concluded that the application of mutation testing in this context is hard and time–consuming. A lot of manual work was required to mutate each class individually, classify its mutants and compute the results of each operation afterward. This was even more challenging when considering reusable classes participating in the implementation of different operations. This introduced two new statuses for mutants, *uncovered* and *partially equivalent* mutants, and this made the processing of results even more difficult. Emerging works studying the application of mutation testing at the system and functional levels are a promising starting point to address this issue [27].

Limitations in current mutation tools. Mutation tools for Java include ExMan [7], Javalanche [15], JavaMut [9], Jester [29], Jumble [40], MuJava [21, 26] and MuClipse [37]. At the time of writing this article, we found that MuJava and its plug-in for Eclipse, MuClipse, were the only publicly available tools that provided support for class–level mutation operators in Java. When used in our study, we found this and other related tools had several practical limitations. In the following, we summarize those features that we missed during our study as well as those that we found more useful and that we consider should be part of any successful mutation tool:

- *Highly automated*. From a user perspective, we missed a major degree of automation in current tools particularly when concerning the computation of results. In addition to basic results such as the mutation score and list of alive and killed mutants, detailed statistics for each operator would be highly desirable. A deep analysis of these statistics would help users in taking decisions like detecting candidate operators to be omitted or selected in future evaluation or refinements of a test suite.

- *Flexible*. Tools should be able to work with real-world applications. Note that this not only means support for a large number of files or lines of code but also support for conventional library formats (e.g. jar files) as well as different versions of Java, features that we missed in our study.

- *Configurable*. Mutation tools should provide configuration utilities and extension points. Adding new mutation operators and result formats should be possible. Filters are also a helpful feature that we missed in our work. During generation, it would have been useful to have filters to specify fragments of code not to be mutated as this would have saved us the time of discarding mutants manually. During execution, our custom filter assisted us in selecting the set of mutants to be considered or omitted. This was especially useful when studying mutant equivalence.

- *Usable*. Mutation tools must be intuitive and easy to use. Visual interfaces (e.g. for examining mutants), configuration wizards and integration with the development environment proved to be helpful features in our study.

- *Automated detection of equivalent mutants*. Although the detection of equivalent mutants is an undecidable problem, some techniques have been proposed to automate it, at least partially [16, 30, 32, 33]. The development, improvement and integration of these techniques in mutation tools is undoubtedly one the main points to make mutation testing affordable in real scenarios.

7. Threats to validity

Our mutation results apply only to three of the reasoners integrated into FaMa framework and therefore may not extrapolate to other programs. Similarly, the number of real faults studied was not large enough to allow us to draw general conclusions. We may remark, however, that 4 out of the 5 real faults studied were found in current releases of two real tools which make them especially motivating for our study. Furthermore, we emphasize that our results may be helpful in supporting and complementing the results obtained in similar studies with OO applications.

The detection of equivalent mutants, an undecidable problem in general, was performed by hand resulting in a tedious and error-prone task. Thus, we must concede a small margin of error in the data regarding equivalence. We remark, however, that these results were taken from three different reasoners providing a fair confidence in the validity of the average data.

8. Related studies

Several researchers have reported the result of experimentation with mutation in OO systems. Kim *et al.* [18] were the first to introduce *Class Mutation* as a means to introduce faults targeting OO specific features in Java. In their work, the authors presented three mutation operators and applied them to three simple classes (Queue, Dequeue and PriorityQueue) resulting in 23 mutants studied. Later, in [19], the authors presented a larger empirical study using mutation testing to evaluate the effectiveness of three different OO testing methods. They applied both traditional (unspecified number) and 15 class–level mutation operators. As subject program, they used a subset of five classes (690 LoC) from the experimental system Product Starter Kit for Java of IBM. Results included the mutation scores for each testing technique under evaluation together with killed rates for each class–level operator. As a final remark, the authors concluded that their work was not extensive and further experimental work would be useful.

Ma *et al.* [26] presented a method to reduce the execution cost of mutation testing. The authors used MuJava to mutate the 266 classes of BCEL, a popular byte code engineering library, and collected data on the number of mutants generated by 24 class–level mutation operators. They also selected seven BCEL classes and studied them in depth comparing the performance of different techniques for the generation and execution of mutants. In a later work [21], the authors conducted two empirical studies to evaluate the usefulness of class–level mutation operators. They applied 5 traditional and 23 class–level mutants to the BCEL system collecting generation statistics for each operator. They also reported some conclusions based on a further study on 11 BCEL classes including killing rates and number of equivalent mutants. Again, the authors conceded that the study was conducted on one sample program and thus the result may not be representative.

Smith *et al.* [36] conducted an empirical study using the MuClipse mutation tool in the back–end of a small Java web–based application, iTrust, developed for academic purposes. In their study, the authors studied the behaviour of mutation operators and tried to identify patterns in the implementation code that remained untested. They applied all 15 traditional and 28 class–level operators available in MuClipse to three of the classes of the iTrust system. Detailed statistics for each operator were reported and analyzed. In a later work [37], the authors extended their empirical study by mutating three of the classes of the open source library HtmlParser 1.6.

Offutt *et al.* [33] studied the number of mutants generated when applying mutation to the 866 classes of six open source Java projects. They applied 29 class–level operators using MuJava mutation tool. Generation statistics were presented for each operator including approximate percentage of equivalent mutants calculated using an automated technique. No results about execution of mutants were provided. Later, in [22], Ma *et al.* extended the work by studying the class-level operators that generated less mutants and concluded that some of them could be eliminated.

Alexander *et al.* [1] proposed an object mutation approach along with a set of mutation operators for inserting faults into objects implementing well–known Java interfaces (e.g. Iterator). To show the feasibility of their approach, they applied their mutation operators to five open source and classroom programs. They mutated, however, only a few locations in one class for each program resulting in 128 mutants studied.

Grün *et al* [15] study the impact of equivalent mutants. For their study, the authors mutated the JAXEN XPATH query engine (12,449 LoC), a popular open source project, resulting in 9,819 mutants. The authors used a custom mutation tool for Java, Javalanche, implementing a small set of sufficient traditional operators as proposed by Offutt [31] and later adapted by Andrews et al. [2]. They selected a subset of 20 mutants and studied them in depth reporting result about the number of equivalent mutants as well as the causes that made them to be equivalent.

Derezinska *et al.* [12] studied the application of mutation testing in C# programs. In their work, the authors applied 5 out of 40 mutation operators proposed for C# to nine available programs with a total size of 298,460 LoC (including comments). Mutation was performed using their tool CREAM (CREAtor of Mutants). As results, generation and execution statistics were given for each mutation operator.

When compared to previous studies, our work contributes to the field of practical experimentation with mutation in different ways. We used a real–world application as subject tool for the mutation instead of using sample or teaching programs [1, 18, 36, 37]. We fully mutated three of the reasoners integrated into FaMa rather than selecting a subset of classes from them [1, 19, 21, 26, 36, 37]. Similarly, we selected a complete set of traditional and class-level operators for Java instead of using a subset of these [12, 15, 18, 26, 33]. Equivalent mutants were detected and examined manually, rather than using partially effective automated mechanisms [22, 33], providing helpful feedback about the detection effort and equivalence causes. In addition to this, we complement our mutation results with the results of some real faults found in the literature and two real tools for the analysis of feature models. Our results include the time invested by our generator to detect each fault which provides interesting information to compare mutants and real bugs. We are not aware of any other work reporting results of real faults in OO programs in the context of mutation. Similarly, we are not aware of any other experience report providing results of mutation applied at a functional level considering testing unit as black-boxes instead of a single class. Finally, to the best of our knowledge, this is the first work reporting the experience with mutation testing from a user perspective, leading to a number of lessons learned helpful for both researcher and practitioner in the field following our steps.

9. Conclusions

This article reports our experience gained from using mutation testing to measure the effectiveness of an automated test data generator for the automated analysis of feature models. We applied both traditional and class–level mutation operators to three of the reasoners integrated into FaMa, an open source Java framework currently used for teaching, research and commercial purposes. We also compared and contrasted our results with the data obtained from some motivating faults found in the literature and recent releases of two real tools, FaMa and SPLOT. A key contribution of our study is that it is reported from a user perspective focusing on how mutation can help when evaluating a test mechanism. Our results are summarized in a number of lessons learned emphasizing important issues concerning the effectiveness and limitations of OO mutation in practice. These may be the seed for further research in the field.

Overall, the mutation scores obtained in our study were supported by the results obtained when detecting real faults in FaMa and SPLOT. This shows the effectiveness of mutation testing in measuring the ability of our generator in detecting faults. Also, the number of mutants generated in our subject classes was much lower than the number of mutants generated in procedural programs supporting the applicability of mutation testing in OO systems. Regarding drawbacks, we found several practical limitations in the current tool support for mutation that hindered our work. We also found that development of techniques and tools for the application of mutation at a system level is a challenging open issue in the mutation testing community.

Material

Our test data generator together with the mutants and test classes used in our evaluation are available at `http://www.lsi.us.es/~segura/files/material/ist10/`. Notice that the access to the standalone

version of SPLOT is restricted to reviewers exclusively following the desires of their authors. The password to access this material is *'splotIST10splot'*.

Acknowledgments

We would like to thank Dr. Macario Polo and the reviewers of the special issue whose comments and suggestions helped us to improve the article substantially. We would also like to thank Dr. Marcilio Mendonca for kindly sending us a standalone version of SPLOT to be used in our evaluation and allowing us to publish the results in benefit of the research community.

This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT project SETI (TIN2009-07366) and the Andalusian Government project ISABEL (TIC-2533).

A. Mutation operators used in our study

Operator	Description
AODS	Arithmetic Operator Deletion (Short-cut)
AODU	Arithmetic Operator Deletion (Unary)
AOIS	Arithmetic Operator Insertion (Short-cut)
AOIU	Arithmetic Operator Insertion (Unary)
AORB	Arithmetic Operator Replacement (Binary)
AORS	Arithmetic Operator Replacement (Short-cut)
ASRS	Assignment Operator Replacement (Short-cut)
COD	Conditional Operator Deletion
COI	Conditional Operator Insertion
COR	Conditional Operator Replacement
LOD	Logical Operator Deletion
LOI	Logical Operator Insertion
LOR	Logical Operator Replacement
ROR	Relational Operator Replacement
SOR	Shift Operator Replacement

Table 14: Traditional mutation operators

References

[1] R. T. Alexander, J.M. Bieman, S. Ghosh, and B. Ji. Mutation of java objects. *International Symposium on Software Reliability Engineering*, 0:341, 2002.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 402–411, New York, NY, USA, 2005. ACM.

[3] D. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines Conference, LNCS 3714*, pages 7–20, 2005.

[4] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615 – 636, 2010.

[5] D. Benavides, P. Trinidad, S. Segura, and A. Ruiz-Cortés. Fama framework. `http://www.isa.us.es/fama/`. accessed May 2010.

[6] D. Le Berre. Sat4j. `http://www.sat4j.org/`. accessed May 2010.

[7] J.S. Bradbury, J.R. Cordy, and J. Dingel. Exman: A generic and customizable framework for experimental mutation analysis. *Mutation Analysis, Workshop on*, 0:4, 2006.

[8] C. Cetina, J. Fons, and V. Pelechano. Moskitt feature modeler. `http://www.pros.upv.es/mfm`.

[9] P. Chevalley and P. Thévenod-Fosse. A mutation analysis tool for java programs. *International Journal on Software Tools for Technology Transfer (STTT)*, 5(1):90–103, November 2003.

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

Language feature	**Operator**	**Description**
Inheritance	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	Overriding method calling position change
	IOR	Overriding method rename
	IPC	Explicit call of a parent's constructor deletion
	ISD	*super* keyword deletion
	ISI	*super* keyword insertion
Overloading	OAC	Arguments of overloading method call change
	OMD	Overloading method deletion
	OMR	Overloading method contents replace
Polimorphism	PCC	Cast type change
	PCD	Type cast operator deletion
	PCI	Type cast operator insertion
	PNC	*new* method call with child class type
	PMD	Member variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PRV	Reference assignment with other comparable variable
Java-specific features	JTI	*this* keyword insertion
	JTD	*this* keyword deletion
	JSI	*static* modifier insertion
	JSD	*static* modifier deletion
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
Common programming mistakes	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Acessor method change
	EMM	Modifier method change

Table 15: Class-level mutation operators

[11] R.A. DeMillo and A.J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.
[12] A. Derezinska and A. Szustek. Tool-supported advanced mutation approach for verification of c# programs. In *Proceedings of the 2008 Third International Conference on Dependability of Computer Systems DepCoS-RELCOMEX*, pages 261–268, Washington, DC, USA, 2008. IEEE Computer Society.
[13] P.G. Frankl, S.N. Weiss, and C. Hu. All-uses vs mutation testing: an experimental comparison of effectiveness. *Journal of Systems and Software*, 38(3):235–253, 1997.
[14] A. Gotlieb and B. Botella. Automated metamorphic testing. In *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*, pages 34 – 40, 3-6 2003.
[15] B. Grün, D. Schuler, and A. Zeller. The impact of equivalent mutants. In *Proceedings of the 4th International Workshop on Mutation Testing*, 2009.
[16] R. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Focus*, 9(4):233–262, 1999.
[17] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. Technical Report TR-09-06, Crest centre, King's college London, 2009.
[18] S. Kim, J.A. Clark, and J.A. Mcdermid. Assessing test set adequacy for object-oriented programs using class mutation. In *Proceedings of the 3rd Symposium on Software Technology (SoST'99)*, pages 72–83, Buenos Aires, Argentina, September 1999.
[19] S. Kim, J.A. Clark, and J.A. Mcdermid. Investigating the effectiveness of object-oriented testing strategies with the mutation method. *Software Testing, Verification and Reliability*, 11:207–225, 2001.
[20] K. Kuchcinski and R. Szymanek. Jacop. `http://jacop.osolpro.com/`. accessed May 2010.
[21] Y.S. Ma, M.J. Harrold, and Y.R. Kwon. Evaluation of mutation testing for object-oriented programs. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 869–872, New York, NY, USA, 2006. ACM.
[22] Y.S. Ma, Y.R. Kwon, and S.W. Kim. Statistical investigation on class mutation operators. *ETRI Journal*, 31:140–150, 2009.
[23] Y.S. Ma, Y.R. Kwon, and J. Offutt. Inter-class mutation operators for java. In *ISSRE '02: Proceedings of the 13th International Symposium on Software Reliability Engineering*, page 352, Washington, DC, USA, 2002. IEEE Computer Society.
[24] Y.S. Ma and J. Offutt. Description of class mutation operators for java, `http://cs.gmu.edu/~offutt/mujava/mutopsClass.pdf`, 2005. accessed May 2010.
[25] Y.S. Ma and J. Offutt. Description of method-level mutation operators for java, `http://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf`, 2005. accessed May 2010.
[26] Y.S. Ma, J. Offutt, and Y.R. Kwon. Mujava: An automated class mutation system. *Software Testesting, Verification and Reliability*, 15(2):97–133, 2005.
[27] P. Reales Mateo, M.P. Usaola, and A.J. Offutt. Mutation at system and functional levels. In *Proceedings of the 5th International Workshop on Mutation Analysis (MUTATION'10)*, Paris, France, 6 April 2010.
[28] M. Mendonca, M. Branco, and D. Cowan. S.P.L.O.T.: Software Product Lines Online Tools. `http://www.splot-research.org/`. accessed May 2010.
[29] I. Moore. Jester. http://jester.sourceforge.net/, 2001. accessed May 2010.
[30] A.J. Offutt and W.M. Craft. Using compiler optimization techniques to detect equivalent mutants. *Journal of Software Testing, Verification, and Reliability*, 4:131–154, 1994.
[31] A.J Offutt, A. Lee, G. Rothermel, R.H Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, 1996.
[32] A.J. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, 7(3):165–192, 1997.
[33] J. Offutt, Y.S. Ma, and Y.R. Kwon. The class-level mutants of mujava. In *AST '06: Proceedings of the 2006 international workshop on Automation of software test*, pages 78–84, New York, NY, USA, 2006. ACM.
[34] Pure-systems. pure::variants. `http://www.pure-systems.com/`. accessed May 2010.
[35] S. Segura, R.M. Hierons, D. Benavides, and A. Ruiz-Cortés. Automated test data generation on the analyses of feature models: A metamorphic testing approach. In *International Conference on Software Testing, Verification and Validation*, pages 35–44, Paris, France, 2010. IEEE press.
[36] B.H. Smith and L. Williams. An empirical evaluation of the mujava mutation operators. *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, 0:193–202, 2007.
[37] B.H. Smith and L. Williams. On guiding the augmentation of an automated test suite via mutation analysis. *Empirical Software Engineering*, 14(3):341–369, 2009.
[38] Sourceforge. Junit. `http://www.junit.org/`. accessed May 2010.
[39] Sourceforge. Metrics, `http://metrics.sourceforge.net/`. accessed May 2010.
[40] Sourceforge. Jumble. http://jumble.sourceforge.net/, 2007. accessed May 2010.
[41] P. Trinidad, D. Benavides, A. Ruiz-Cortés, S. Segura, and A.Jimenez. Fama framework. In *12th Software Product Lines Conference (SPLC)*, 2008.
[42] J. Whaley. Javabdd. `http://javabdd.sourceforge.net/`. accessed May 2010.