Automated Test Data Generation on the Analyses of Feature Models: A Metamorphic Testing Approach

Sergio Segura*, Robert M. Hierons†, David Benavides* and Antonio Ruiz-Cortés*
*Department of Computer Languages and Systems, University of Seville
Av Reina Mercedes S/N, 41012 Seville, Spain
†School of Information Systems, Computing and Mathematics, Brunel University
Uxbridge, Middlesex, UB7 7NU United Kingdom

Abstract

A Feature Model (FM) is a compact representation of all the products of a software product line. The automated extraction of information from FMs is a thriving research topic involving a number of analysis operations, algorithms, paradigms and tools. Implementing these operations is far from trivial and easily leads to errors and defects in analysis solutions. Current testing methods in this context mainly rely on the ability of the tester to decide whether the output of an analysis is correct. However, this is acknowledged to be time-consuming, error-prone and in most cases infeasible due to the combinatorial complexity of the analyses.

In this paper, we present a set of relations (so-called metamorphic relations) between input FMs and their set of products and a test data generator relying on them. Given an FM and its known set of products, a set of neighbour FMs together with their corresponding set of products are automatically generated and used for testing different analyses. Complex FMs representing millions of products can be efficiently created applying this process iteratively. The evaluation of our approach using mutation testing as well as real faults and tools reveals that most faults can be automatically detected within a few seconds.

1. Introduction

Software Product Line (SPL) engineering is a systematic approach to develop families of software products. Products in SPLs are defined in terms of features. A *feature* is an increment in product functionality [2]. *Feature models* [15] are widely used to represent all the valid combinations of features (i.e. products) of an SPL in a single model. The automated analysis of feature models deals with the computer–aided extraction of information from feature models. Typical operations of analysis allow determining whether a feature model is void (i.e. it represents no products), whether it contains errors (e.g. features that cannot be part of any product) or what is the number of products of the SPL represented by the model. Catalogues with a number of analysis operations identified on feature models are reported in the literature [4], [5], [19].

Many approaches have been proposed to automate the analysis of feature models. Most translate feature models into logic paradigms such as propositional logic [1], [10], [14], [17], [21] description logic [13], [24] or constraint programming [3], [22]. Others propose ad-hoc algorithms and solutions to perform these analyses [12], [18]. Additionally, these analysis capabilities can also be found in both commercial and open source tools such as *AHEAD Tool Suite*[1], *FaMa Framework*[2], *Feature Model Plug-in*[3] and *pure::variants*[4].

Feature model analysis tools commonly deal with complex data structures and algorithms. This makes analyses far from trivial and easily leads to errors increasing development time and reducing reliability of analysis solutions. Current testing methods in this context mainly rely on the ability of the tester to decide whether the output of an analysis is correct. However, this is recognized to be time–consuming, error–prone and in most cases infeasible due to the combinatorial complexity of the analyses. This limitation, also found in many other software testing domains, is known as the *oracle problem* [25] i.e. impossibility to determine the correctness of a test output.

Metamorphic testing [7], [25] was proposed as a way to address the oracle problem. The idea behind this technique is to generate new test cases based on existing test data. The expected output of the new test cases can be checked by using known relations (so–called *metamorphic relations*) among two or more input data and their expected outputs. Key benefits of this technique are that it does not require an oracle and it can be highly automated.

In this paper, we propose using metamorphic testing for the automated generation of test data for the analyses of feature models. In particular, we present a set of metamorphic

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)

1. http://www.cs.utexas.edu/users/schwartz/ATS.html
2. http://www.isa.us.es/fama/
3. http://gp.uwaterloo.ca/fmp/
4. http://www.pure-systems.com/

relations between feature models and their set of products and a test data generator relying on them. Given a feature model and its known set of products, our tool generates a set of neighbour models together with their associated sets of products. Complex feature models representing million of products can be efficiently generated by applying this process iteratively. Once generated, products are automatically inspected to get the expected output of a number of analyses over the models. A key benefit of our approach is that it is highly generic being suitable to test any operation extracting information from the set of products of a feature model. In order to show the feasibility of our approach, we evaluated the ability of our test data generator to detect faults in three different scenarios, namely:

- **Mutation testing.** We introduced a number of artificial faults into three of the analysis reasoners integrated into the FaMa framework and checked the effectiveness of our generator to detect them. As a result, we got an overall mutation score of over 98% in the three reasoners with average detection times under 7.5 seconds.
- **A real fault.** We developed a mock tool including a motivating fault found in the literature and checked the ability of our approach to detect it automatically. As a result, the fault was detected in all the operations tested with a score of 91.6% and an average detection time of 20.2 seconds.
- **A real tool.** We finally evaluated our approach with a recent release of the FaMa Framework, *FaMa v1.0.0 alpha*, detecting two defects.

The remainder of the paper is structured as follows: Section 2 presents feature models, their analyses and metamorphic testing in a nutshell. A detailed description of our metamorphic relations and test data generator is presented in Section 3. Section 4 describes the evaluation of our approach in different scenarios. Finally, we summarize our conclusions and describe our future work in Section 5.

2. Preliminaries

2.1. Feature models

A *feature model* defines the valid combination of features in a domain. A feature model is visually represented as a tree–like structure in which nodes represent features, and edges illustrate the relationships among them. Figure 1 shows a simplified example of a feature model representing an e–commerce SPL. The model illustrates how features are used to specify and build on–line shopping systems. The software of each application is determined by the features that it provides. The root feature (i.e. E-Shop) identifies the SPL.

Feature models were first introduced in 1990 as a part of the FODA (Feature–Oriented Domain Analysis) method [15] as a means to represent the commonalities and variabilities of system families. Since then, feature modelling has been widely adopted by the software product line community and a number of extensions have been proposed in attempts to improve properties such as succinctness and naturalness [19]. Nevertheless, there seems to be a consensus that at a minimum feature models should be able to represent the following relationships among features:

- **Mandatory.** If a child feature is mandatory, it is included in all products in which its parent feature appears. For instance, every on–line shopping system in our example must implement a *catalogue* of products.
- **Optional.** If a child feature is defined as optional, it can be optionally included in products in which its parent feature appears. For instance, *banners* is defined as an optional feature.
- **Alternative.** A set of child features are defined as alternative if only one feature can be selected when its parent feature is part of the product. In our SPL, a shopping system has to implement *high* or *medium* security policy but not both in the same product.
- **Or-Relation.** A set of child features are said to have an or-relation with their parent when one or more of them can be included in the products in which its parent feature appears. A shopping system can implement several payment modules: *bank draft*, *credit card* or both of them.

Notice that a child feature can only appear in a product if its parent feature does. The root feature is a part of all the products within the SPL. In addition to the parental relationships between features, a feature model can also contain cross-tree constraints between features. These are typically of the form:

- **Requires.** If a feature A requires a feature B, the inclusion of A in a product implies the inclusion of B in such product. On–line shopping systems accepting payments with *credit card* must implement a *high* security policy.
- **Excludes.** If a feature A excludes a feature B, both features cannot be part of the same product. Shopping systems implementing a *mobile* GUI must not include support for *banners*.

2.2. Automated analysis of feature models

The automated analysis of feature models deals with the computer–aided extraction of information from feature models. From the information obtained, marketing strategies and technical decisions can be derived. Catalogues with a number of analysis operations identified on feature models are reported in the literature [4], [5], [19]. Next, we summarize some of the analysis operations we will refer to through the rest of the paper.

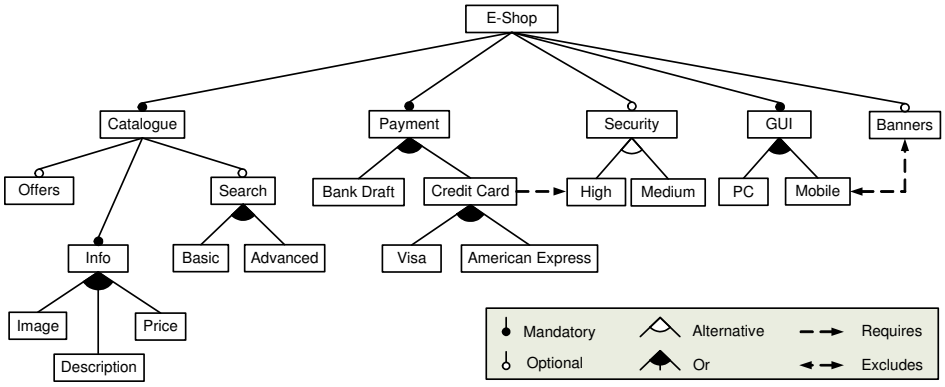

Figure 1. A sample feature model

- **Determining if a feature model is void.** This operation takes a feature model as input and returns a value stating whether the feature model is void or not. A feature model is *void* if it represents no products. [1], [3], [12], [13], [14], [17], [18], [19], [21], [24].
- **Finding out if a product is valid.** This operation checks whether an input product (i.e. set of features) belongs to the set of products represented by a given feature model or not. As an example, let us consider the feature model of Figure 1 and the following product *P*={*E-Shop, Catalogue, Info, Description, Security, Medium, GUI, PC*}. Notice that P is not a valid product of the product line represented by the model because it does not include the mandatory feature *'Payment'*. [1], [3], [14], [17], [18], [19], [21], [24].
- **Obtaining all products.** This operation takes a feature model as input and returns all the products represented by the model. [1], [3], [12], [14], [21].
- **Calculating the number of products.** This operation returns the number of products represented by a feature model. The model in Figure 1 represents 2016 different products. [3], [12], [17].
- **Calculating variability.** This operation takes a feature model as input and returns the ratio between the number of products and $2^n - 1$ where n is the number of features in the model [3]. This operation may be used to measure the flexibility of the product line. For instance, a small factor means that the number of combinations of features is very limited compared to the total number of potential products. In Figure 1, *Variability* = 0.00048.
- **Calculating commonality.** This operation takes a feature model and a feature as inputs and returns a value representing the proportion of valid products in which the feature appears [3]. This operation may be used to prioritize the order in which the features are to be developed and can also be used to detect dead features [22]. In Figure 1, *Commonality(Banners)* = 25%.

Previous operations can be performed automatically using different approaches. Most translate feature models into specific logic paradigms such as propositional logic [1], [14], [17], [21] description logic [13], [24] or constraint programming [3], [22]. Others propose ad-hoc algorithms and solutions to perform these analyses [12], [18]. Finally, previous analysis capabilities can also be found in several commercial and open source tools such as *AHEAD Tool Suite*, *FaMa Framework*, *Feature Model Plug-in* and *pure::variants*.

2.3. Metamorphic testing

An *oracle* in software testing is a procedure by which testers can decide whether the output of a program is correct [25]. In some situations, the oracle is not available or it is too difficult to apply. This limitation is referred in the testing literature as the *oracle problem* [26]. Consider, as an example, checking the results of complicated numerical computations or the processing or non-trivial outputs like the code generated by a compiler. Furthermore, even when the oracle is available, the manual prediction and comparison of the results are in most cases time–consuming and error–prone.

Metamorphic testing [7], [25] was proposed as a way to address the oracle problem. The idea behind this technique is to generate new test cases based on existing test data. The expected output of the new test cases can be checked by using so–called *metamorphic relations*, that is, known relations among two or more input data and their expected outputs. As a positive result of this technique, there is no need for an oracle and the testing process can be highly automated.

Consider, as an example, a program that compute the cosine function ($cos(x)$). Suppose the program produces output -0.3999 when run with input $x = 42$ radians. An important property of the cosine function is $cos(x) = cos(-x)$. Using this property as a metamorphic relation, we could

design a new test case with $x = -42$. Assume the output of the program for this input is 0.4235. When comparing both outputs, we could easily conclude the program is not correct.

Metamorphic testing has shown to be effective in a number of testing domains including numerical programs [8], graph theory [9] or service–oriented applications [6].

3. Our approach

3.1. Metamorphic relations on feature models

In this section, we define a set of metamorphic relations between feature models and their corresponding set of products. We relate feature models using the concept of neighbourhood. Given a feature model, FM, we say that FM' is a *neighbour model* if it can be derived from FM by adding or removing a relationship or constraint R. The metamorphic relations between the products of a model and the one of their neighbours are then determined by R as follows:

Mandatory. Consider the models and associated set of products depicted in Figure 2. FM' is created from FM by adding a mandatory feature ('D') to it. The semantics of mandatory relationships state that mandatory features must always be part of the products in which is parent feature appears. Based on this, we conclude that the set of expected products of *FM'* is correct iff it preserves the set of products of *FM* and extends it by adding the new mandatory feature,'D', in all the products including its parent feature,'A'. In the example, therefore, this relation is fulfilled.

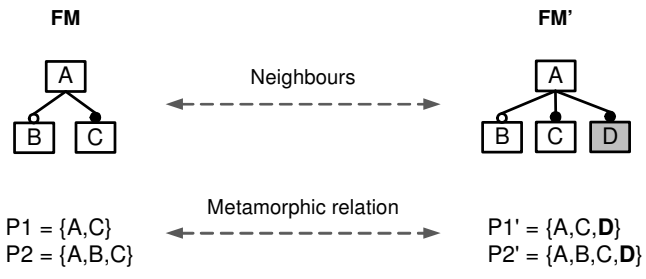

Figure 2. Neighbour model. Mandatory feature added

Formally, let f be the mandatory feature added to the model and pf its parent feature. Consider the functions $prods(FM)$, returning the set of products of an input feature models, and $features(P)$, returning the set of features of a given product. We use the symbol '#' to refer to the cardinality (i.e. number of elements) of a set. We define the relation between the set of products of FM and the one of FM' as follows:

$$\begin{aligned}&\#prods(FM') = \#prods(FM) \wedge \\ &\forall P'(P' \in prods(FM') \Leftrightarrow \exists P \in prods(FM) \cdot \\ &(pf \in features(P) \wedge P' = P \cup \{f\}) \vee \\ &(pf \notin features(P) \wedge P' = P))\end{aligned} \tag{1}$$

Optional. Let f be the optional feature added to the model and pf its parent feature. Consider the function $filter(FM, S, E)$ that returns the set of products of FM including the features of S and excluding the features of E. The metamorphic relation between FM and FM' is defined as follows:

$$\begin{aligned}&\#prods(FM') = \#prods(FM) + \#filter(FM, \{pf\}, \varnothing) \wedge \\ &\forall P'(P' \in prods(FM') \Leftrightarrow \exists P \in prods(FM) \cdot \\ &P' = P \vee (pf \in features(P) \wedge P' = P \cup \{f\}))\end{aligned} \tag{2}$$

Alternative. Let C be the set of alternative subfeatures added to the model and pf their parent feature. The relation between the set of products of FM and FM' is defined as follows:

$$\begin{aligned}&\#prods(FM') = \#prods(FM) + (\#C - 1)\#filter(FM, \{pf\}, \varnothing) \wedge \\ &\forall P'(P' \in prods(FM') \Leftrightarrow \exists P \in prods(FM) \cdot \\ &(pf \in features(P) \wedge \exists c \in C \cdot P' = P \cup \{c\}) \vee \\ &(pf \notin features(P) \wedge P' = P))\end{aligned} \tag{3}$$

Or. Let C be the set of subfeatures added to the model and pf their parent feature. We denote with $\wp(C)$ the powerset of C i.e. the set of all subsets in C. This metamorphic relation is defined as follows:

$$\begin{aligned}&\#prods(FM') = \#prods(FM) + (2^{\#C} - 2)\#filter(FM, \{pf\}, \varnothing) \wedge \\ &\forall P'(P' \in prods(FM') \Leftrightarrow \exists P \in prods(FM) \cdot \\ &(pf \in features(P) \wedge \exists S \in \wp(C) \cdot P' = P \cup S) \vee \\ &(pf \notin features(P) \wedge P' = P)))\end{aligned} \tag{4}$$

Requires. Let f and g be the origin and destination features of the new requires constraint added to the model. The relation between the set of products of FM and FM' is defined as follows:

$$prods(FM') = prods(FM) \setminus filter(FM, \{f\}, \{g\}) \tag{5}$$

Excludes. Let f and g be the origin and destination features of the new excludes constraint added to the model. This metamorphic relation is defined as follows:

$$prods(FM') = prods(FM) \setminus filter(FM, \{f, g\}, \varnothing) \tag{6}$$

3.2. Automated test data generation

The semantics of a feature model is defined by the set of products that it represents [19]. Most analysis operations on feature models can be answered by inspecting this set adequately. Based on this, we propose a two–step process to

automatically generate test data for the analyses of feature models as follows:

Feature model generation. We propose using previous metamorphic relations together with model transformations to generate feature models and their respective set of products. Note that this is a singular application of metamorphic testing. Instead of using metamorphic relations to check the output of different computations, we use them to actually compute the output of follow–up test cases. Figure 3 illustrates an example of our approach. The process starts with an input feature model whose set of products is known. A number of step–wise transformations are then applied to the model. Each transformation produces a neighbour model as well as its corresponding set of products according to the metamorphic relations. Transformations can be applied either randomly or using heuristics. This process is repeated until a feature model (and corresponding set of products) with the desired properties is generated.

Test data extraction. Once a feature model with the desired properties is generated, it is used as non-trivial input for the analysis. Similarly, its set of products is automatically inspected to get the output of a number of analysis operations i.e. any operation that extracts information from the set of products of the model. As an example, consider the model and set of products generated in Figure 3 and the analysis operations described in Section 2.2. We can obtain the expected output of all of them by simply answering the following questions:

- *Is the model void?* No, the set of products is not empty.
- *Is P={A,C,F} a valid product?* Yes. It is included in the set.
- *How many different products represent the model?* 6 different products.
- *What is the variability of the model?* $6/2^7 - 1 = 0.047$
- *What is the commonality of feature* B*?* Feature B is included in 4 out of the 6 products of the set. Therefore its commonality is 66.6%

We may remark that we could have also used a 'pure' metamorphic approach, start with a known feature model, transform this to obtain a neighbour model, and use metamorphic relations to check the outputs of the tool under test. However, this strategy would force us to define metamorphic relations for each operation meanwhile our approach can be used generically to generate test data for any analysis that extracts information from the set of products. Key benefit of our approach is that it can be easily automated enabling the generation and execution of test cases without the need for a human oracle.

3.3. A prototype tool

As a part of our proposal, we implemented a prototype tool relying on our metamorphic relations. The tool receives a feature model and its associated set of products as input and returns a modified version of the model and its expected set of products as output. If no inputs are specified, a new model is generated from scratch.

Our prototype applies random transformations to the input model increasing its size progressively. The set of products is efficiently computed after each transformation according to the metamorphic relations presented in Section 3.1. Transformations are performed according to a number of parameters including number of features, percentage of constraints, maximum number of subfeatures on a relationship and percentage of each type of relationship to be generated.

The number of products of a feature model increases exponentially with the number of features. This was a challenge during the development of our tool causing frequent time deadlocks and memory overflows. To overcome these problems, we optimized our implementation using efficient data structures (e.g. boolean arrays) and limiting the number of products of the models generated. Using this setup, feature models with up to 11 million products were generated in a standard laptop machine within a few seconds.

The tool was developed on top of FaMa Benchmarking System v0.7 (FaMa BS)[5]. This system provides a number of capabilities for benchmarking in the context of feature models including random generators as well as readers and writers for different formats. Figure 4 depicts a random feature model generated with our prototype tool and exported from FaMa BS to the graph visualization tool GraphViz[6]. The model has 20 features and 20% of constraints. Its set of products contains 22,832 different feature combinations.

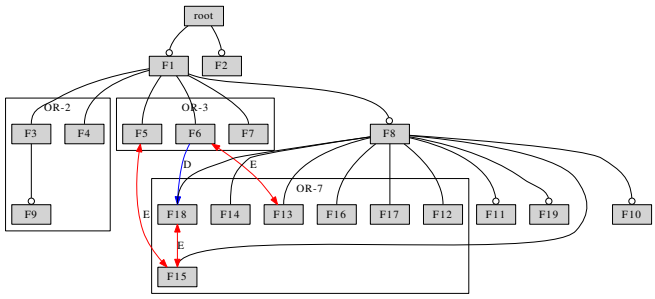

Figure 4. Input feature model generated with our tool

4. Evaluation

4.1. Evaluation using mutation testing

In order to measure the effectiveness of our proposal, we evaluated the ability of our test data generator to detect faults in the software under test (i.e. so–called fault-based

5. http://www.isa.us.es/fama/?FaMa_Benchmarking
6. http://www.graphviz.org/

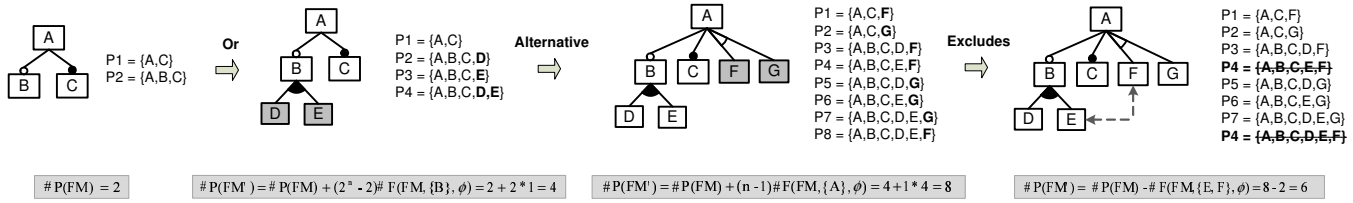

Figure 3. An example of random feature model generation using metamorphic relations

adequacy criterion). To that purpose, we applied mutation testing on an open source framework for the analysis of feature models.

Mutation testing [11] is a common fault–based testing technique that measures the effectiveness of test cases. Briefly, the method works as follows. First, simple faults are introduced in a program creating a collection of faulty versions, called *mutants*. The mutants are created from the original program by applying syntactic changes to its source code. Each syntactic change is determined by a so–called *mutation operator*. Test cases are then used to check whether the mutants and the original program produce different responses. If a test case distinguishes the original program from a mutant we say the mutant has been *killed* and the test case has proved to be effective at finding faults in the program. Otherwise, the mutant remains *alive*. Mutants that keep the program's semantics unchanged and thus cannot be detected are referred to as *equivalent*. The percentage of killed mutants with respect to the total number of them (discarding equivalent mutants) provides an adequacy measurement of the test suite called *mutation score*.

4.1.1. Experimental setup. We selected FaMa Framework as a good candidate to be mutated. FaMa is an open source framework integrating different reasoners for the automated analysis of feature models and currently being integrated into the commercial tools MOSKitt[7] and pure::variants[8]. As creators of the tool, it was feasible for us to use it for the mutations. In particular, we selected three reasoners integrated into the framework, namely: Sat4jReasoner v0.9.2 (using satisfiability problems by means of Sat4j[9] solver), JavaBDDReasoner v0.9.2 (using binary decision diagrams by means of JavaBDD[10] solver) and JaCoPReasoner v0.8.3 (using constraint programming by means of JaCoP[11] solver). Each one 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, the six analysis operations presented in Section 2.2 were tested.

7. http://www.moskitt.org
8. In the context of the DiVA European project (http://www.ict-diva.eu/)
9. http://www.sat4j.org/
10. http://javabdd.sourceforge.net/
11. http://jacop.osolpro.com/

To automate the mutation process, we used MuClipse Eclipse plug-in v1.3[12]. MuClipse is a Java visual tool for mutation testing based on MuJava [16]. It supports a wide variety of operators and can be used for both generating mutants and executing them in separated steps. Despite this, we still found several limitation in the tool. On the one hand, the current version of MuClipse 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. On the other hand, we found the execution component provided by this and other related tools not flexible enough, providing as a result mainly mutation score and list of alive and killed mutants. To address our needs, we developed a custom execution module providing some extra functionality including: 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) filtering capability to specify which mutants should be considered or ignored during the execution.

Test cases were generated randomly using our prototype tool as described in Section 3.2. In the cases of operations receiving additional inputs apart from the feature model (e.g. valid product), these were selected using a basic partition equivalence strategy. For each operation, test cases with the desired properties were generated and run until a fault was found or a timeout was exceeded. Feature models were generated with an initial size of 10 features and 10% (with respect to the number of features) of constraints for efficiency. This size was then incremented progressively according to a configurable increasing factor. This factor was typically set to 10% and 1% (every 20 test cases generated) for features and constraints respectively. The maximum size of the set of products was equally limited for efficiency. This was configured according to the complexity of each operation and the performance of each reasoner with typical values of 2000, 5000 and 11000000. For the evaluation of our approach, we followed three steps, namely:

1) *Reasoners testing.* Prior to their analysis, we checked whether the original reasoner passed all the tests. A timeout of 60 seconds was used. As a result, we detected and fixed a defect affecting the computation

12. http://muclipse.sourceforge.net/

of the set of products in JaCoPReasoner. We found this fault to be especially motivating since it was also present in the current release of FaMa (see Section 4.2 for details).

2) *Mutants generation.* We applied all the traditional mutation operators available in MuClipse, a total of 15. Specific mutation operators for object–oriented code were discarded to keep the number of mutants manageable. For details about these operators we refer the reader to [16].
3) *Mutants execution.* For each mutant, we ran our test data generator and tried to find a test case that kills it. An initial timeout of 60 seconds was set for each execution. This timeout was then repeatedly incremented by 60 seconds (until a maximum of 600) with remaining alive mutants recorded. Equivalent mutants were manually identified and discarded after each execution.

Both the generation and execution of mutants was performed in 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.

4.1.2. Analysis of results. Table 1 shows information about the size of the reasoners and the number of generated mutants. Lines of code (LoC) do not include lines in blank and comments. Out of the 749 generated mutants, 94 of them (i.e. 13.4%) were identified as semantically equivalent. In addition to these, we manually discarded 87 mutants (i.e. 11.6%) affecting secondary functionality of the subject programs (e.g. computation of statistics) not addressed by our current test data generator.

Tables 2, 3 and 4 show the results of the mutation process on Sat4jReasoner, JavaBDDReasoner and JaCoPReasoner respectively. For each operation, the number of classes involved, number of executed mutants, test data generation results and mutation score are presented. Test data results include average and maximum time required to kill each mutant, average and maximum number of test cases generated to kill a mutant and maximum timeout that showed to be effective in killing any mutant, i.e. further increments in the timeout did not kill any new mutant.

Note that the functionality of each operation was scattered in several classes. Some of these were reusable being used in more than one operation. Mutants on these reusable classes were evaluated separately with the test data of each operation using them for more accurate mutation scores. This explains why the number of executed mutants on each reasoner (detailed in Tables 2, 3 and 4) is higher that the number of mutants generated for that reasoner (showed in Table 1).

Reasoner	LoC	Mutants	Equivalent	Discarded
Sat4jReasoner	743	262	27	47
JavaBDDReasoner	625	302	28	37
JaCoPReasoner	686	185	46	3
Total	**2054**	**749**	**101**	**87**

Table 1. Mutants generation results

Results revealed an overall mutation score of over 98% in the three reasoners. Operation *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. In this time, 414 different test cases were generated and run. This shows the efficiency of the generation process. The maximum timeouts required to kill a mutant were 600 seconds for the operation *VoidFM*, 120 for the operation *ValidProduct* and 60 seconds for the rest of analyses. This gives an idea of the minimum timeout that should be used when applying our approach in real scenarios.

Figure 5 depicts a spread graph with the size (number of features and constraints) of the feature models that killed mutants in the operation *VoidFM*. As illustrated, small feature models were in most cases sufficient to find faults. This was also the trend in the rest of the operations. This suggests that the procedure used for the generation of models, starting from smaller and moving progressively to bigger ones, is adequate and efficient.

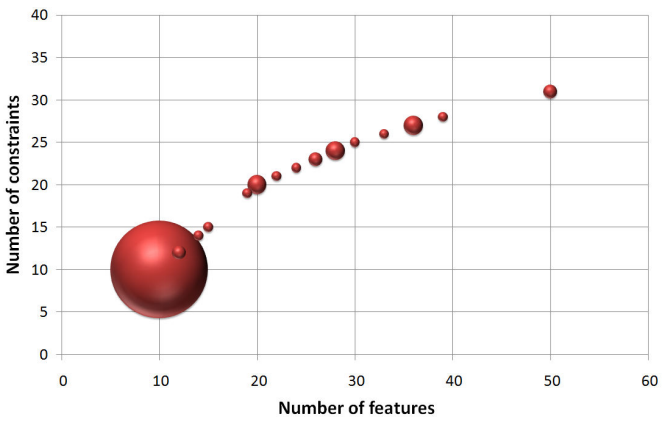

Figure 5. Feature models killing mutants in the operation *VoidFM*

Operations		Executed Mutants		Test Data Generation					Score
Name	Classes	Total	Alive	Av Time (s)	Max time (s)	Av TCs	Max TCs	Timeout (s)	
VoidFM	2	55	0	37.6	566.5	95.1	414	600	100
ValidProduct	5	109	3	4.3	88.6	12	305	120	97.2
Products	2	86	0	0.6	3.4	1.5	12	60	100
#Products	2	57	0	0.7	2.4	1.8	8	60	100
Variability	3	82	0	0.6	1.7	1.3	5	60	100
Commonality	5	109	0	0.6	3.8	1.5	13	60	100
Total	**19**	**498**	**3**	**7.4**	**566.5**	**18.9**	**414**		**99.3**

Table 2. 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	Timeout (s)	
VoidFM	2	75	3	6.6	111.7	29.3	350	120	96
ValidProduct	5	129	5	1	34.6	3.8	207	60	96.1
Products	2	130	0	0.7	34.6	1.4	12	60	100
#Products	2	77	0	0.5	1.4	1.6	6	60	100
Variability	3	104	0	0.5	2.4	1.6	12	60	100
Commonality	5	131	0	0.5	3	1.5	16	60	100
Total	**19**	**646**	**8**	**1.6**	**111.7**	**6.5**	**350**		**98.7**

Table 3. 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	Timeout (s)	
VoidFM	2	8	0	1.5	8.3	11.3	83	60	100
ValidProduct	5	61	0	0.7	1.2	1.3	5	60	100
Products	2	37	0	0.5	0.7	1	1	60	100
#Products	2	13	0	0.5	0.7	1	1	60	100
Variability	3	36	0	0.5	0.7	1	1	60	100
Commonality	5	66	0	0.5	0.7	1.1	3	60	100
Total	**19**	**221**	**0**	**0.7**	**8.3**	**2.8**	**83**		**100**

Table 4. Test data generation results using traditional operators in JaCoPReasoner

Operation	Av Time (s)	Max Time (s)	Av TCs	Max TCs	Timeout (s)	Score
VoidFM	78.2	229.1	515.8	905	600	100
ValidProduct	38.4	43.7	268.4	322	600	50
Products	1.1	2.9	5.7	19	60	100
#Products	1.0	2.7	5.4	16	60	100
Variability	1.2	2.1	6.4	13	60	100
Commonality	1.4	3	7.8	20	60	100
Total	**20.2**	**229.1**	**134.9**	**905**		**91.6**

Table 5. Evaluation results using a motivating fault reported in the literature

Finally, we may mention that experimentation with Sat4jReasoner revealed a serious defect affecting its scalability. The reasoner created a temporary file for each execution but it did not delete it afterward. We found that the more temporary files were created, the slower become the creation of new ones with delays of up to 30 seconds in the executions of operations. Once detected, the defect was fixed and the experiments repeated. This suggests that our approach could also be applicable to scalability testing.

4.2. Evaluation using real tools and faults

4.2.1. A motivating fault. Consider the work of Batory in SPLC'05 [1], one of the seminal papers in the community of automated analysis of feature models. The paper included a bug (later fixed[13]) in the mapping of a feature model to a propositional formula. We implemented this wrong mapping into a mock reasoner for FaMa using JavaBDD and checked the effectiveness of our approach in detecting the fault.

Figure 6 illustrates an example of the wrong output caused by the fault. This manifests itself in alternative relationships whose parent feature is not mandatory making reasoners to consider as valid product those including multiple alternative subfeatures (P3). As a result, the set of products returned by the tool is erroneously larger than the actual one. For instance, the number of products returned by our faulty tool when using the model in Figure 1 as input is 3584 (instead of the actual 2016). Note that this is a motivating fault since it can easily remain undetected even when using an input with the problematic pattern. Hence, in the previous example (either with *'security'* feature as mandatory or optional), the mock tool correctly identifies the model as non void (i.e. it represents at least one product), and so the fault remains latent.

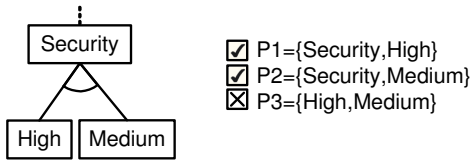

Figure 6. Wrong set of products obtained with the faulty reasoner

Table 5 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. The results are based on 10 executions. The fault was detected in all the executions performed in 5 out of 6 operations. Average and maximum times were higher than the ones obtained when using mutants but still low being 229.1 seconds (3.8 minutes) in the worst case. The fault remained latent in the 50% of the executions performed in the *ValidProduct* operation. When examining the data, we concluded that this was due to the basic strategies used for the selection of inputs products for this operation. We presume that using more complex heuristic for this purpose would improve the results.

13. ftp://ftp.cs.utexas.edu/pub/predator/splc05.pdf

4.2.2. FaMa v1.0.0 alpha. Finally, we evaluated our tool by trying to detect faults in a recent release of the FaMa Framework, *FaMa v1.0.0 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.

5. Conclusions and future work

In this paper, we presented a set of metamorphic relations on feature models and an automated test data generator relying on them. Given a feature model and its set of products, our tool generates neighbouring models and their corresponding set of products. Generated products are then inspected to obtain the expected output of a number of analysis over the models. Non-trivial feature models representing millions of products can be efficiently generated applying this process iteratively. In order to evaluate our approach, we checked the effectiveness of our tool to detect faults using mutation testing as well as real faults and tools. Two defects were detected in a recent release of FaMa, an open source framework currently being integrated into several commercial tools. Our results show that the application of metamorphic testing on the domain of automated analysis of feature models is efficient and effective detecting most faults in a few seconds without the need of a human oracle.

We identify several challenges for our future work in two main directions:

- **Address more operations.** A wide number of analysis operations on feature models focus on detecting anomalies in the models such as redundancies [23] or dead features [22]. We plan to extend our tool for generating test data for these operations. To this purpose, we intend to design heuristics for the generation of input feature models containing different types of inconsistencies.
- **Combination with other testing strategies.** In our current approach, feature models are created from scratch for simplicity. However, it is known that metamorphic testing produces better results when combined with other test case selection strategies that generate the

initial set of test cases. We are currently working in the design of this set of test cases using black-box testing techniques. A preliminary version of this test suite is available in [20].

Our prototype tool together with the mutants and test classes used in our evaluation are available at http://www.lsi.us.es/~segura/files/material/icst10/.

References

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

[2] D. Batory, D. Benavides, and A. Ruiz-Cortés. Automated analysis of feature models: Challenges ahead. *Communications of the ACM*, December:45–47, 2006.

[3] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. *LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, 3520:491–503, 2005.

[4] D. Benavides, A. Ruiz-Cortés, P. Trinidad, and S. Segura. A survey on the automated analyses of feature models. In *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, 2006.

[5] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models: A detailed literature review. Technical Report ISA-09-TR-04, ISA research group, 2009. Available at http://www.isa.us.es/.

[6] W. Chan, S. Cheung, and K. Leung. A metamorphic testing approach for online testing of service-oriented software applications. *International Journal of Web Services Research*, 4(2):61–81, 2007.

[7] T.Y. Chen, S.C. Cheung, and S.M. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, University of Science and Technology, Hong Kong, 1998.

[8] T.Y. Chen, J. Feng, and T.H. Tse. Metamorphic testing of programs on partial differential equations: a case study. In *Proceedings of the 26th International Computer Software and Applications Conference*, pages 327–333, 2002.

[9] T.Y. Chen, D.H. Huang, T.H. Tse, and Z.Q. Zhou. Case studies on the selection of useful relations in metamorphic testing. In *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004)*, pages 569–583, 2004.

[10] K. Czarnecki and P. Kim. Cardinality-based feature modeling and constraints: A progress report. In *Proceedings of the International Workshop on Software Factories At OOPSLA*, 2005.

[11] 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.

[12] A. van Deursen and P. Klint. Domain–specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.

[13] S. Fan and N. Zhang. Feature model based on description logics. In *Knowledge-Based Intelligent Information and Engineering Systems*, pages 1144–1151. 2006.

[14] Rohit Gheyi, Tiago Massoni, and Paulo Borba. A theory for feature models in alloy. In *First Alloy Workshop*, pages 71–80, Portland, United States, nov 2006.

[15] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature–Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, 1990.

[16] Y. Ma, J. Offutt, and Y. Kwon. Mujava: a mutation system for java. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 827–830, New York, NY, USA, 2006. ACM.

[17] M. Mannion. Using first-order logic for product line model validation. In *Proceedings of the Second Software Product Line Conference (SPLC2)*, LNCS 2379, pages 176–187, San Diego, CA, 2002. Springer.

[18] X. Peng, W. Zhao, Y. Xue, and Y. Wu. Ontology-based feature modeling and application-oriented tailoring. In *ICSR*, pages 87–100, 2006.

[19] P. Schobbens, P. Heymans, J. Trigaux, and Y. Bontemps. Feature Diagrams: A Survey and A Formal Semantics. In *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06)*, Minneapolis, Minnesota, USA, September 2006.

[20] S. Segura, D. Benavides, and A. Ruiz-Cortés. FaMa Test Suite v1.1. Technical Report ISA-09-TR-03, ISA research group, 2009. Available at http://www.isa.us.es/.

[21] J. Sun, H. Zhang, Y.F. Li, and H. Wang. Formal semantics and verification for feature modeling. In *Proceedings of the ICECSS05*, 2005.

[22] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software*, 81(6):883–896, 2008.

[23] T. von der Massen and H. Lichter. Deficiencies in feature models. In Tomi Mannisto and Jan Bosch, editors, *Workshop on Software Variability Management for Product Derivation - Towards Tool Support*, 2004.

[24] H. Wang, Y. Li, J. Sun, H. Zhang, and J. Pan. A semantic web approach to feature modeling and verification. In *Workshop on Semantic Web Enabled Software Engineering*, 2005.

[25] E.J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[26] Z.Q. Zhou, DH. Huang, TH. Tse, Z. Yang, H. Huang, and TY. Chen. Metamorphic testing and its applications. In *Proceedings of the 8th International Symposium on Future Software Technology*, pages 346–351, 2004.