Automated Metamorphic Testing on the Analyses of Feature Models☆

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. A Feature Model (FM) represents the valid combinations of features in a domain. The automated extraction of information from FMs is a complex task that involves numerous analysis operations, techniques and tools. Current testing methods in this context are manual and 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, this is known as the *oracle problem*.
Objective. In this paper, we propose using metamorphic testing to automate the generation of test data for feature model analysis tools overcoming the oracle problem. An automated test data generator is presented and evaluated to show the feasibility of our approach.
Method. We present a set of relations (so-called metamorphic relations) between input FMs and the set of products they represent. Based on these relations and given a FM and its known set of products, a set of neighbouring FMs together with their corresponding set of products are automatically generated and used for testing multiple analyses. Complex FMs representing millions of products can be efficiently created by applying this process iteratively.
Results. Our evaluation results using mutation testing and real faults reveal that most faults can be automatically detected within a few seconds. Two defects were found in FaMa and another two in SPLOT, two real tools for the automate analysis of feature models. Also, we show how our generator outperforms a related manual suite for the automated analysis of feature models and how this suite can be used to guide the automated generation of test cases obtaining important gains in efficiency.
Conclusion. Our results show that the application of metamorphic testing in the domain of automated analysis of feature models is efficient and effective in detecting most faults in a few seconds without the need for a human oracle.

Key words: Metamorphic testing, test data generation, mutation testing, feature models, automated analysis, product lines.

1. Introduction

Software Product Line (SPL) engineering is a reuse strategy to develop families of related systems [19]. From common assets, different software products are assembled reducing production costs and time–to–market. Products in SPLs are defined in terms of features. A *feature* is an increment in product functionality [3]. *Feature models* [32] are widely used to represent all the valid combinations of features (i.e. products) of an SPL in a single model in terms of features and relations among them (see Figure 1).

The automated analysis of feature models deals with the computer–aided extraction of information from feature models [5]. Typical operations of analysis allow determining whether a feature model is

☆A preliminary version of this paper was presented in [47]

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

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 up to 30 analysis operations on feature models have been reported [5, 44]. Analysis solutions can be mainly categorized into those using propositional logic [2, 20, 26, 34, 36, 55, 61], constraint programming [4, 52, 59], description logic [23, 57] and adhoc algorithms [24, 54, 56]. Additionally, there are both commercial and open source tools supporting these analysis capabilities such as *AHEAD Tool Suite* [1], *Big Lever Software Gears* [7], *FaMa Framework* [22], *Feature Model Plug-in* [25], *pure::variants* [42] and *SPLOT* [35, 51].

Feature model analysis tools deal with complex data structures and algorithms (FaMa framework contains over 20 000 lines of code). This makes the implementation of analyses far from trivial and easily leads to errors increasing development time and reducing reliability of analysis solutions. Gaining confidence in the absence of faults in these tools is especially relevant since the information extracted from feature models is used all along the SPL development process to support both marketing and technical decisions [3]. Thus, the lack of specific testing mechanisms in this context appears as a major obstacle for engineers when trying to assess the functionality and quality of their programs.

In [45, 46], we gave a first step to address the problem of functional testing on the analyses of feature models. In particular, we presented a set of manually designed test cases, so-called FaMa Test Suite (FaMa TeS), to validate the implementation of the analyses on feature models. Although effective, we found several limitations in our manual approach that motivated this work. First, evaluation results with artificial and real faults showed room for improvement in terms of efficacy. Second, the manual design of new test cases relied on the ability of the tester to decide whether the output of an analysis was correct. We found this was time–consuming, error–prone and in most cases infeasible due to the combinatorial complexity of the analyses. As a result, we were force to use small and in most cases oversimplistic input models whose output could be calculated by hand. This limitation, also found in many other software testing domains, is known as the *oracle problem* [58] i.e. impossibility to determine the correctness of a test output.

Metamorphic testing [12, 58] 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 relations between feature models and their set of products and a test data generator based on them. Given a feature model and its known set of products, our tool generates a set of neighbouring 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. Key benefits of our approach are that it removes the oracle problem and 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 main scenarios. First, we introduced hundreds of artificial faults (i.e. mutants) into three of the analysis components integrated into the FaMa framework (hereafter referred to as *reasoners*) and checked the effectiveness of our generator to detect them. As a result, our automated test data generator found more than 98.5% of the faults in the three reasoners with average detection times under 7.5 seconds. Second, 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 91.4% and an average detection time of 23.5 seconds. Finally, we evaluated our approach with recent releases of two real tools for the analysis of feature models, FaMa and SPLOT, detecting two defects in each of them.

This article extends our previous work on automated test data generation for the analyses of feature models [47] in several ways. First, we show how our generator can be used to automatically test the detection of dead features in feature models (i.e. those that cannot be selected). Second, we explain how we evaluated our approach by trying to find faults in SPLOT, a real on-line tool for the automated

analysis of feature models, finding two bugs on it. Third, we show how our automated test data generator outperforms our manual suite for the analyses of feature models by experimental results with both mutants and real faults. Finally, we present a refined version of our generator using the manual test cases of FaMa TeS as an initial test set to guide the generation of follow-up test cases. Experimental results reveal that refining our approach in this way lead to important gains in efficiency.

The rest of the article is structured as follows: Section 2 presents feature models, their analyses and metamorphic testing. 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 as well as the comparison with FaMa TeS. We show how our approach can be refined by combining it with other test case selection strategies in Section 5. Section 6 discusses the main threats to validity of our work. In Section 7, we present the related works in the field of metamorphic testing and compare them with our approach. Finally, we summarize our conclusions in Section 8.

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 [32] 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 [44]. 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, *offers* 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 this 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 cannot include support for *banners*.

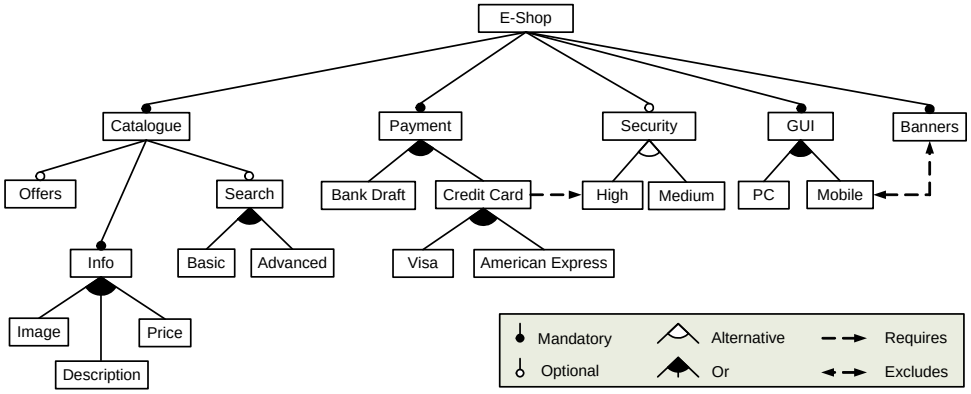


Figure 1: A sample feature model

A generalization of the classical notation presented in this section are the so-called *cardinality–based* feature models [20]. In this notation, alternative and or–relations are replaced by a so-called *group cardinality* of the form $[n..n']$, with n as lower bound and n' as upper bound limiting the number of child features that can be part of a product. Hence, a set relationship with a group cardinality [1..1] is equivalent to an alternative relationship while a group cardinality of [1..N], being N the number of children of the set relationship, is equivalent to an or-relation.

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 up to 30 analysis operations identified on feature models are reported in the literature [5, 44]. Next, we summarize some of the analysis operations we will refer to through the rest of the article.

- **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. [2, 4, 20, 23, 26, 34, 36, 44, 53, 54, 55, 56, 57, 61].

- **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, Banners}*. 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'*. [2, 4, 20, 26, 34, 44, 53, 57, 59].

- **Obtaining all products.** This operation takes a feature model as input and returns all the products represented by the model. A feature model is void if the set of products that it represents is empty. [2, 4, 26, 34, 53, 54, 56].

- **Calculating the number of products.** This operation returns the number of products represented by a feature model. The model in Figure 1 represents 504 different products. [4, 20, 24, 34, 53, 54, 56].

- **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 [4, 53]. 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* = 0.00012.

- **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 [4, 24, 53]. 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 [52]. In Figure 1, *Commonality(Search)* = 75%.

- **Detecting dead features.** This operation takes a feature model as input and returns the set of dead features included in the model. A feature is *dead* if it cannot appear in any of the products derived from the model. Dead features are caused by a wrong usage of cross-tree constraints and are clearly undesired since they give a wrong idea of the domain. As an example, note that features *'Mobile'* and *'Banners'* in Figure 1 are mutually exclusive. However, Figure *'Banners'* is mandatory and must be included in all the products of the product lines. This means that feature *'Mobile'* can never be selected and therefore is dead. [3, 20, 36, 52, 53, 54, 61].

These operations can be performed automatically using different approaches. Most translate feature models into specific logic paradigms such as propositional logic [2, 20, 26, 34, 36, 55, 61], constraint programming [4, 52, 59] or description logic [23, 57]. Others propose ad-hoc algorithms and solutions to perform these analyses [24, 54, 56]. Finally, these analysis capabilities can also be found in several commercial and open source tools such as *AHEAD Tool Suite* [1], *Big Lever Software Gears* [7], *FaMa Framework* [22], *Feature Model Plug-in* [25], *pure::variants* [42] and *SPLOT* [35, 51].

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 [58]. In some situations, the oracle is not available or it is too difficult to apply. This limitation is referred to in the testing literature as the *oracle problem* [62]. Consider, as an example, checking the results of complicated numerical computations or the processing of 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 [12, 58] 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 [13], graph theory [14] or service–oriented applications [8].

3. Automated Metamorphic Testing on the Analyses of Feature Models

3.1. Metamorphic Relations on Feature Models

In this section, we define a set of metamorphic relations between feature models (i.e. input) and their corresponding set of products (i.e. output). These metamorphic relations are derived from the basic operators of feature models, that is, the different types of relationships and constraints among features. In particular, 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 neighbours models and associated set of products depicted in Figure 2. FM' in Figure 2(a) is created from FM by adding a mandatory feature ('D') to it, i.e. they are neighbours. The

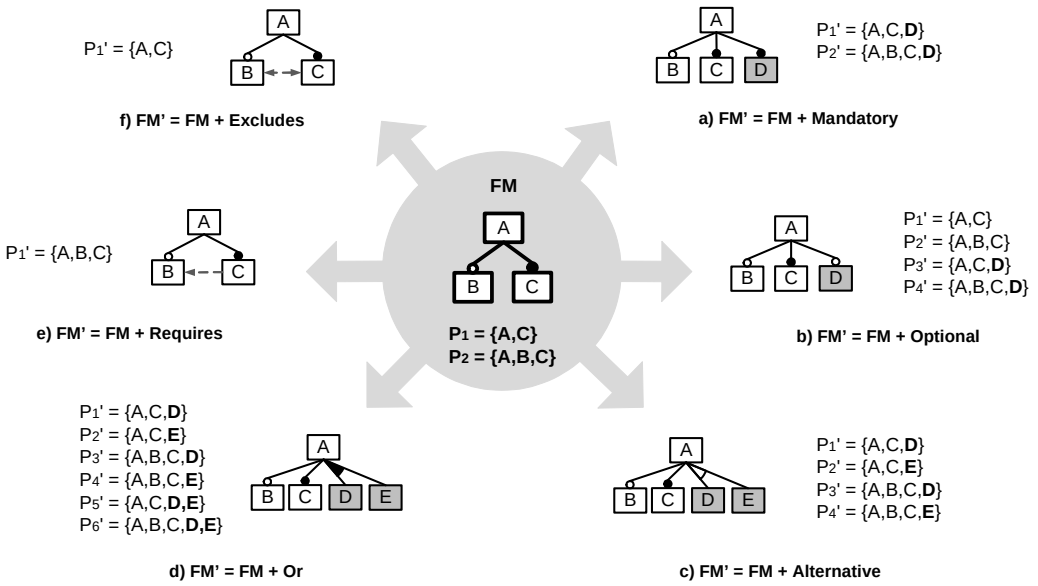


Figure 2: Some examples of neighbour feature models

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 incorrect if it does not preserve 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. Formally, let f be the mandatory feature added to the model and pf its parent feature, 'D' and 'A' in the example respectively. Consider the functions $products(FM)$, 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:

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

Optional. Let f be the optional feature added to the model and pf its parent feature. An example is presented in Figure 2(b) with $f = D$ and $pf = A$. 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 the set of products of FM and that of FM' is defined as follows:

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

Alternative. Let C be the set of alternative subfeatures added to the model and pf their parent feature. In Figure 2(c), $C = \{D, E\}$ and $pf = A$. The relation between the set of products of FM and FM' is defined as follows:

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

Or. Let C be the set of subfeatures added to the model and pf their parent feature. For instance, in Figure 2(d), $C = \{D, E\}$ and $pf = A$. 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:

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

Requires. Let f and g be the origin and destination features of the new requires constraint added to the model. In Figure 2(e), $f = C$ and $g = B$. The relation between the set of products of FM and FM' is defined as follows:

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

Excludes. Let f and g be the origin and destination features of the new excludes constraint added to the model. This is illustrated in Figure 2(f) with $f = B$ and $g = C$. This metamorphic relation is defined as follows:

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

3.2. Automated Test Data Generation

The semantics of a feature model is defined by the set of products that it represents [44]. 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 (e.g. number of features) is generated.

Test data extraction. Once a feature model with the desired properties is created, 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?* $6/(2^9 - 1) = 0.011$
- *What is the commonality of feature* B*?* Feature B is included in 5 out of the 6 products of the set. Therefore its commonality is 83.3%

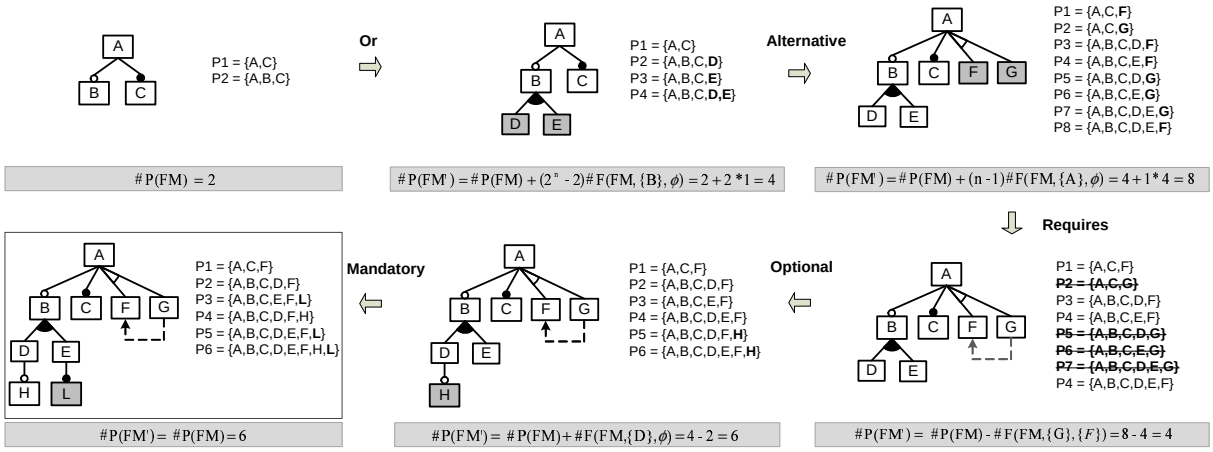


Figure 3: An example of random feature model generation using metamorphic relations

- *Does the model contain any dead feature?* Yes. Feature G is dead since it is not included in any of the products represented by the model.

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 require to define metamorphic relations for each operation. In contrast, we propose to use the metamorphic relations to compute the output of follow-up test cases instead of simply comparing the results of different tests. Starting from a trivial test case, we can generate increasingly larger and more complex test cases making sure that the metamorphic relations are fulfilled at each step. This allows us to define the metamorphic relations for a single operation, *Products*, from which we derive the expected output of many of the other analyses on feature models. A 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.

Finally, we would like to emphasize that the operations presented are only some examples of the analyses that can be tested using our approach. We estimate that this technique could be used to test, at least, 16 out of the 30 analysis operations identified in [5]. The operations out of the scope of our approach are mainly those looking for specific patterns in the feature tree.

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 limited 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) [22]. 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 [28]. The model has 20 features and 20% of constraints. Its set of products contains 22,832 different feature combinations.

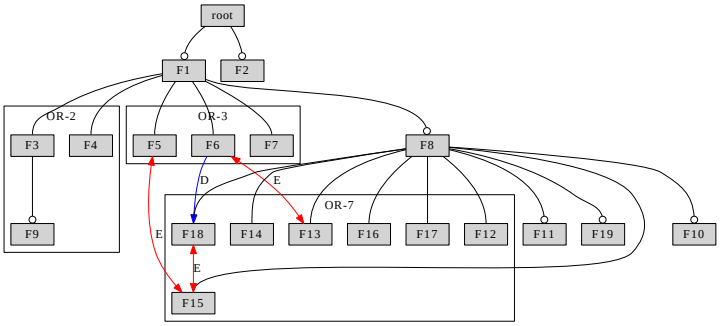


Figure 4: Sample input feature model generated with our tool

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

Mutation testing [21] 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 the *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 is currently being integrated into the commercial tools MOSKitt [37] and pure::variants[1]. As creators of FaMa, it was feasible for us to use it for the mutations. In particular, we selected three of the analysis components integrated into the framework (so-called reasoners), namely: Sat4jReasoner v0.9.2 (using satisfiability problems by means of Sat4j solver [43]), JavaBDDReasoner v0.9.2 (using binary decision diagrams by means of JavaBDD solver [31]) and JaCoPReasoner v0.8.3 (using constraint programming by means of JaCoP solver [30]). 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 seven analysis operations presented in Section 2.2 were tested. The operation *DeadFeatures*, however, was tested in JaCoPReasoner exclusively since it was the only reasoner implementing it.

[1]In the context of the DiVA European project (http://www.ict-diva.eu/)

To automate the mutation process, we used MuClipse Eclipse plug-in v1.3 [50]. MuClipse is a Java visual tool for mutation testing based on MuJava [33]. 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 to not be sufficiently flexible, providing as a result mainly mutation score and lists 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), the additional inputs 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 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 [33].
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 blank lines and comments. Out of the 760 generated mutants, 103 of them (i.e. 13.5%) were identified as semantically equivalent. In addition to these, we manually discarded 87 mutants (i.e. 11.4%) 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 (until the maximum of 600s) did not kill any new mutant.

Note that the functionality of each operation was scattered in several classes. Some of these were 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 than the number of mutants generated for that reasoner (showed in Table 1).

Results revealed an overall mutation score of over 98.5% in the three reasoners. 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. Similarly, the operation *DeadFeatures* revealed a mutation score of 100% in JaCoPReasoner with an average number of test cases of 2.3. 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 required for these operations were noticeable higher than for the other analysis operations 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 analysis operations. This gives an idea of the minimum timeout that should be used when applying our approach in other 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 means that feature models with an initial size of 10 features and 10% of cross-tree constraints were complex enough to exercise most of the features of the analysis reasoners under test. This suggests that the procedure used for the generation of models, starting from smaller and moving progressively to bigger ones, is adequate and efficient.

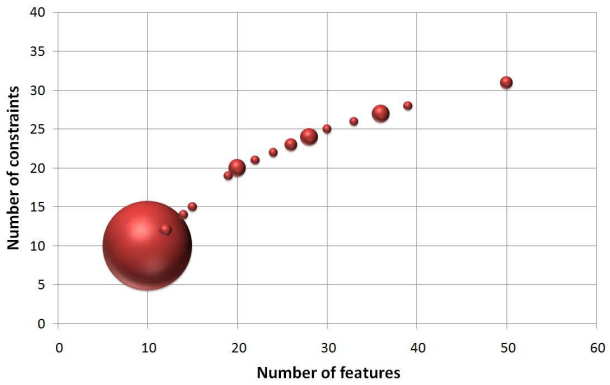


Figure 5: Size of the feature models killing mutants in the operation *VoidFM*

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 became 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

Reasoner	**LoC**	**Mutants**	**Equivalent**	**Discarded**
Sat4jReasoner	743	262	27	47
JavaBDDReasoner	625	302	28	37
JaCoPReasoner	791	196	48	3
Total	**2159**	**760**	**103**	**87**

Table 1: Mutants generation results

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**		**99.4**

Table 2: Test data generation results 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 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
DeadFeatures	5	80	0	0.8	2.1	2.3	14	60	100
Total	**24**	**301**	**0**	**0.7**	**8.3**	**2.7**	**83**		**100**

Table 4: Test data generation results in JaCoPReasoner

experiments repeated. This suggests that our approach could also be applicable to scalability testing.

For more details about the evaluation of our approach using mutation testing we refer the reader to [48, 49].

4.2. Evaluation using Real Tools and Faults

4.2.1. A Motivating Fault

Consider the work of Batory in SPLC'05 [2], one of the seminal papers in the community of automated analysis of feature models. The paper included a bug (later fixed[2]) in the mapping of a feature model to a propositional formula. We implemented this wrong mapping into a mock reasoner for FaMa using the CSP-based solver Choco [18] 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 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 896 (instead of the actual 504). 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.

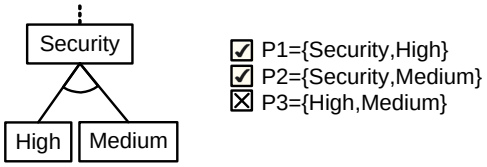


Figure 6: Wrong set of products obtained with the faulty reasoner

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 6 out of 7 operations. Most of the average and maximum times were higher than the ones obtained when using mutants but still low being 191.9 seconds (3.2 minutes) in the worst case. The fault remained latent in 40% 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.

Operation	**Av Time (s)**	**Max Time (s)**	**Av TCs**	**Max TCs**	**Score**
VoidFM	101.2	191.9	294.6	366	100
ValidProduct	41.6	91.8	146.8	312	40
Products	1.8	4.6	4.5	14	100
#Products	2.9	7.9	9.0	28	100
Variability	2.2	3.2	6.1	10	100
Commonality	2.1	4.8	5.6	15	100
DeadFeatures	12.8	29.2	42.3	101	100
Total	**23.5**	**191.9**	**72.7**	**366**	**91.4**

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

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

4.2.2. 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. For each operation, we ran our test data generator 10 times. Tests revealed two defects in all the executions (see Table 6). 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. This fault was detected in 142.9 seconds on average. The second fault, detected in less than one second in all executions, 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. Both bugs were fixed in the new version of the tool.

Operation	**Av Time (s)**	**Max Time (s)**	**Av TCs**	**Max TCs**	**Score**
JaCoP-Products	142.9	198.6	437.3	605	100
Sat4j-ValidProduct	0.6	0.7	1	1	100
Sat4j-Commonality	0.6	0.6	1	1	100
Total	**48**	**198.6**	**146.4**	**605**	**100**

Table 6: Evaluation results with FaMa

4.2.3. SPLOT

Software Product Lines On-line Tools (SPLOT) [35, 51] 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[3] 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 (see Table 7). The first one, detected in less than one second on average, 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, detected in about 0.5 seconds in all cases, 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. Faults detected in the standalone version of the tool were also observed in the online version of SPLOT. We may remark that the authors confirmed the results and told us that they were aware of these limitations.

Operation	**Av Time (s)**	**Max Time (s)**	**Av TCs**	**Max TCs**	**Score**
Sat4j-VoidFM	0.7	1.3	26.7	66	100
Sat4j-#Products	1	2	26.1	66	100
Sat4j-DeadFeatures	0.9	2.2	38.3	134	100
JavaBDD-VoidFM	0.4	0.5	1.5	2	100
JavaBDD-#Products	0.4	0.5	1.9	5	100
Total	**0.7**	**2.2**	**18.9**	**134**	**100**

Table 7: Evaluation results with SPLOT

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

4.3. Comparison with a Manual Test Suite

In this section, we compare the effectiveness of our automated test data generator and FaMa Test Suite, a set of manually designed test cases to test the implementation of analysis operations on feature models. FaMa Test Suite (FaMa TeS) [45, 46] was presented by the authors as a first contribution on the testing of feature model analysis tools. It consists of 180 test cases covering the 7 analysis operations presented in Section 2.2. For its design, we used several black-box testing techniques [41] (e.g. equivalence partitioning) to assist us in the creation of a representative set of input–output combinations. To the best of our knowledge, this is the only available test suite for the analyses of feature models.

Table 8 shows two of the test cases included in FaMa TeS. For each test case, an ID, description, inputs, expected outputs and intercase dependencies (if any) are presented. Intercase dependencies refer to identifiers of test cases that must be executed prior to a given test case [29]. Each test case was designed to reveal a single type of fault. As illustrated, we used trivially small input models so that we could calculate the expected output by hand. This limitation was one of the main motivations that led us to develop the automated metamorphic approach presented in this article.

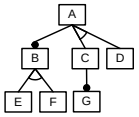
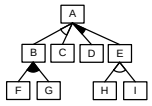
ID	Description	Input	Expected Output	Deps
P-9	Check whether the interaction between mandatory and alternative relationships is correctly processed.		{A,B,D,F}, {A,B,D,E}, {A,B,C,F,G}, {A,B,C,E,G}	P-1 P-4
VP-37	Check whether valid products (with a maximum set of features) are correctly identified in feature models containing or- and alternative relationships.	 P={A,B,D,E,F,G,H}	Valid	VP-5 VP-6 VP-7 VP-8 VP-9 VP-10

Table 8: Two of the test cases included in FaMa Test Suite

In order to enable the objective comparison of our generator and the manual suite, we evaluated FaMa TeS with the same mutants and real faults presented in previous sections. A full summary of the results together with a detailed description of the suite are available in [45] (technical report of 55 pages).

Table 9 depicts the results obtained when using FaMa TeS to kill the mutants in the FaMa reasoners. For each reasoner and operation, the total number of executed mutants, alive mutants and mutation score are presented. On the one hand, all mutants in JaCoPReasoner were killed by the manual suite equalling the results obtained with our metamorphic approach. On the other hand, mutation scores in Sat4jReasoner (94.4%) and JavaBDDReasoner (95.8%) were significantly lower than those obtained with our test data generator (99.4% and 98.7% respectively). This inferiority of the manual suite was also observed in the results of the evaluation with the bugs found in FaMa, SPLOT and the faulty reasoner (i.e. that including the motivating fault found in [2]). These results are depicted in Table 10. In the faulty reasoner, our automated test data generator detected the fault in all the operations meanwhile our manual suite failed to detect the defect in the operations *ValidProduct* and *DeadFeatures*. Similarly, the manual suite was unable to reveal the failure in the operation *Products* of JaCoPReasoner in FaMa 1.0.

From the results obtained and our experience working with FaMa TeS, we conclude that our automated metamorphic approach outperformed the manual suite in multiple ways. First, our automated generator was more effective than the manual suite, i.e. it detected more faults. Second, our metamorphic approach is highly generic so it can easily be adapted to test most analysis operation while the development of manual test cases is tedious and time-consuming. Also, manual test cases are trivially small while our current approach allows the efficient generation of large feature models representing million of products. Finally, and more important, our generator automatically checks the output of tests, removing the oracle problem found when using manual means. All these pieces of evidence support the effectiveness of our

Operation	Sat4jReasoner			JavaBDDReasoner			JaCoPReasoner		
	Mutants	**Alive**	**Score**	**Mutants**	**Alive**	**Score**	**Mutants**	**Alive**	**Score**
VoidFM	55	20	63.6	75	12	84.0	8	0	100
ValidProduct	109	4	96.3	129	7	94.6	61	0	100
Products	86	1	98.8	130	2	98.5	37	0	100
#Products	57	1	98.2	77	2	97.4	13	0	100
Variability	82	1	98.8	104	2	98.1	36	0	100
Commonality	109	1	99.1	131	2	98.5	66	0	100
DeadFeatures	-	-	-	-	-	-	80	0	100
Total	**498**	**28**	**94.4**	**646**	**27**	**95.8**	**301**	**0**	**100**

Table 9: Mutants execution results of the manual test suite

Fault	Automated Generator	Manual Test Suite
Faulty reasoner		
VoidFM	+	+
ValidProduct	+	-
Products	+	+
#Products	+	+
Variability	+	+
Commonality	+	+
DeadFeatures	+	-
Faults in FaMa and SPLOT		
FaMa-JaCoPProducts	+	-
FaMa-Sat4j	+	+
SPLOT-Sat4j	+	+
SPLOT-JavaBDD	+	+

Table 10: Real faults detected by our test data generator and the manual suite

approach when compared to related testing mechanisms for feature model analysis tools in general, and manual mechanisms in particular.

5. Refinement

In the approach presented previously, test cases are randomly generated 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 [12, 13]. In this section, we propose refining our approach by using an initial set of input models that seed the generation of follow-up test cases. This initial set of models could guide the generator to search in specific error-prone areas improving the detection results. To show the feasibility of the proposal, we used the input models in FaMa TeS as seed for the automated generation of test data. Later, we repeated the evaluation with mutants and real faults and checked how the input test cases had contributed to improve the efficiency and effectiveness of our automated generator.

As a preliminary step, we refined our manual suite by adding new test cases that kill the remaining alive mutants found during the evaluation with mutation (see Section 4.3). Notice that this is a natural step when using mutation to improve the quality of the test suite [50]. In order to avoid the suite being overfitted for the mutants under evaluation, we used the information provided by only one of the reasoners that was later excluded for the evaluation. In particular, we selected Sat4jReasoner since it was the one in which more mutants remained alive and therefore the one providing more feedback to improve our suite (see Table 9). As a result, 13 new test cases were added to the manual suite (from 180 to 193), i.e. those that killed the remaining alive mutants in Sat4jReasoner.

Figure 7 illustrates the steps we followed to use the input models of the refined manual suite to guide the generation of follow–up test cases. For each operation, the input models used in their associated test cases in FaMa TeS and their corresponding set of products (calculated manually) are saved (step 1). Then, for each test case to be generated, a feature model is selected (step 2) and extended (step 3) by applying

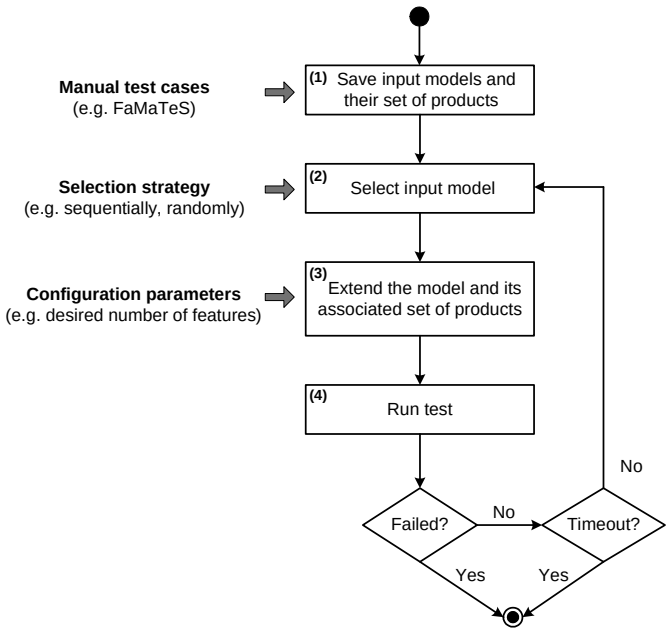


Figure 7: Algorithm for the generation of test cases using a starting manual test suite

a set of step-wise random transformations to it. Each transformation produces a neighbour model as well as its corresponding set of products according to the metamorphic relations presented in Section 3.1. Once a feature model with the desired properties has been generated, the test case is run (step 4) and the execution stopped if a failure is revealed. Otherwise, a new input model from FaMa TeS is selected and the previous process repeated. In our current approach, initial input models are selected sequentially, however, other strategies (e.g. random selection) would also be feasible. A maximum timeout of 600 seconds was used for all the executions. The configuration parameters for the generation (e.g. desired number of features, increasing size factor, etc.) were set to the same values described in Section 4.1.1.

Table 11 depicts the mutants execution results of our refined generator. For each reasoner, the average detection time, maximum detection time, average number of test cases generated and mutation scores are presented. The last row shows the average values in the form x / y where x is the value obtained when using our initial approach (i.e. test cases are created randomly from scratch) and y is the value obtained when using the refined version of our generator (i.e. input models from FaMa TeS are used to guide the generation of test cases). As illustrated, the experiments revealed a significant improvement in the detection times and number of test cases generated before killing a mutant. In JavaBDDReasoner, for instance, the average detection time was reduced by 43.7% (from 1.6 to 0.9 seconds) and the number of test cases was reduced by 63% (from 6.5 to 2.4 test cases). This improvement was especially significant in the maximum detection times reduced by 63.9% (from 111.7 to 40.3 seconds) in JavaBDDReasoner and 79.5% (from 8.3 to 1.7 seconds) in JaCoPReasoner. We may mention that we found some cases, those with lowest times, in which our refined generator was slightly slower than our original approach. As expected, this was caused by the overhead introduced in the new program when loading the initial test set from XML files. Finally, we also found a slight improvement in the mutation score of JavaBDDReasoner, from 98.7% to 98.9%.

The evaluation results with real faults, shown in Table 12, were similar to those obtained with mutants. The average detection times, for instance, were reduced by 41.7% (from 23.5 to 13.7 seconds) in the faulty reasoner and by 43.9% (from 36.2 to 20.3 seconds) in the real faults founds in FaMa and SPLOT. Results in the operation *VoidFM* of our faulty reasoner were especially positive with a reduction in the average detection time of 93.6%, from 101.2 seconds (see Table 5) to 6.4. The mutation score in the operation *ValidProduct* showed no improvement. Again, we think this is due to the basic strategies used for the selection of input products for this operation. More complex heuristic for this purpose could certainly

Operation	JavaBDDReasoner				JaCoPReasoner			
	Av Time (s)	**Max Time (s)**	**Av TCs**	**Score**	**Av Time (s)**	**Max Time (s)**	**Av TCs**	**Score**
VoidFM	1.5	25.7	5.8	97.3	0.8	1.7	2.3	100
ValidProduct	0.9	7.2	2.3	96.1	0.8	1.2	1.3	100
Products	1.0	40.3	1.5	100	0.8	1.1	1.0	100
#Products	0.7	1.5	1.5	100	0.9	1.1	1.1	100
Variability	0.7	3.5	1.6	100	0.8	0.9	1.0	100
Commonality	0.6	2.9	1.4	100	0.8	1.2	1.1	100
DeadFeatures	-	-	-	-	0.8	1.1	1.1	100
Total	1.6 / **0.9**	111.7 / **40.3**	6.5 / **2.4**	98.7 / **98.9**	0.7 / **0.8**	8.3 / **1.7**	2.7 / **1.3**	100 / **100**

Table 11: Mutants execution results of our refined automated test data generator

yield better results. Finally, we may mention that the results obtained in the operation *DeadFeatures* of the faulty reasoner were much worse that those found in our original approach with an average detection time increasing from 12.8 seconds (see Table 5) to 41.3. Interestingly, it seems that starting the generation with models that already had some dead features affected negatively the detection of the fault.

Fault	**Av Time (s)**	**Av TCs**	**Score**
Faulty reasoner			
VoidFM	6.4	22	100
ValidProduct	39.1	145.8	40
Products	2.0	4.7	100
#Products	2.3	5.2	100
Variability	2.0	4.4	100
Commonality	2.9	7.1	100
DeadFeatures	41.3	151.9	100
Total	23.5 / **13.7**	72.7 / **48.7**	91.4 / **91.4**
Faults in FaMa and SPLOT			
FaMa-JaCoPProducts	79.2	244.0	100
FaMa-Sat4j	1.0	1.2	100
SPLOT-Sat4j	0.5	8.7	100
SPLOT-JavaBDD	0.4	1.9	100
Total	36.2 / **20.3**	117.6 / **63.9**	100 / **100**

Table 12: Evaluation results of our refined generator using real faults

These results support the feasibility of combining our test data generator with other testing strategies that generate the initial set of models for a more effective search of faults. However, while the improvement in detection times were noticeable, we may remark that we did not obtain significant improvements in terms of efficacy. Therefore, we encourage researchers and practitioners following our approach to assess carefully the trade–off between the effort required to develop an initial set of test cases and the expected gains in efficiency.

6. Threats to Validity

We briefly discuss the threats to validity of our work.

- **Subject reasoners**. Our mutation results apply only to three of the reasoners integrated into FaMa framework and therefore could not extrapolate to other programs. Nevertheless, we may remark that each one of these reasoners use a different technique to automate the analysis and were coded by different developers providing the required level of heterogeneity for our evaluation.

- **Equivalent mutants**. 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 results were taken from three different reasoners providing a fair confidence in the validity of the average data. Furthermore, equivalence results were also confirmed by the results obtained by our manual suite.

- **Real faults**. The number of real faults in our study was not large enough to allow us to draw general conclusions. However, we may emphasize that these were collected from both the literature and real tools providing a sufficient degree of representativeness. These faults were harder to detect than mutants in general and provided a good idea of the behaviour of our approach in real scenarios.

7. Related Work

The related works in the field of metamorphic testing can be divided into three areas, namely:

Applications. Chen et al. [13] studied the application of metamorphic testing to address the oracle problem in numerical programs. A case study with partial equation was presented. Zhou et al. [62] presented several uses of metamorphic testing in the domains of graph theory, computer graphics, compilers and interactive software. Some metamorphic relations were proposed but no experimental results were reported. Later, in [14], the authors proposed a guideline for the selection of good metamorphic relations and presented two cases studies with the shortest path program and the critical path program. Experimental results of the evaluation of the metamorphic relations using manual mutation testing was reported. In [9], Chan et al. presented a metamorphic approach for integration testing in context–sensitive middleware–based applications. The authors identified functional relations that associate different execution sequences of a test case. Then, they used metamorphic testing to check the results of the test cases and find contradiction on those relations. Chan et al. [8] proposed an approach for online service testing and presented an experiment with a service-oriented calculator of arithmetic expressions to show the feasibility of their work. Chen et al. [11] proposed using metamorphic testing to test bioinformatic programs and presented experimental results with two of those programs.

Tools, frameworks and methods. Gotlieb and Botella [27] proposed an automated testing framework able to check metamorphic relations using constraint programming. Given a program and a metamorphic relation, their tool tries to find test data that violates the relation. Evaluation results with mutation testing were presented. Chan et al. [10] proposed a testing methodology for service-oriented applications based on metamorphic testing. The authors introduced the concept of metamorphic service. A *metamorphic service* is a service that calls the relevant services of the application and check the metamorphic relations. Beydeda [6] proposed a method to enable self-testability of components using metamorphic testing. Murphy et al. [40] presented an extension to the Java Modeling Language (JML) and a tool able to process it. This extension allow users to specify metamorphic relations as annotation in the Java code. These annotation are later processed by their tool that generates test code that can be executed using JML runtime assertion checking, for ensuring that the specifications hold during program execution. Later, in [39], the authors presented a framework called Amsterdam to support metamorphic testing at the system level. They also presented an approach called *Heuristic Metamorphic Testing* to reduce false positives and address some cases of non-determinism. The authors extended their work in [38] presenting a new technique called *Metamorphic Runtime Checking*, a testing approach that automatically conducts metamorphic testing of individual functions during the program's execution. The authors also presented a framework called columbus and presented experimental results.

Integration of metamorphic testing with other testing techniques. Chen et al. [16] proposed a semi–proving method based on metamorphic testing and global symbolic evaluation. The proposed method verifies expected necessary properties for program correctness and identify failure-causing inputs if such properties are not satisfied. Later, in [17], the authors presented an integrated method that combined metamorphic testing and fault–based testing by means of mutation testing. Chen et al. [15] proposed using metamorphic testing in combination with special values testing. Special test values are test values in which their expected results are well known and can be used to verify the program. Some examples with numerical programs were presented. Xie et al. [60] extend the spectrum–based fault localization method with metamorphic testing making it applicable to applications without a test oracle.

When compared to previous studies, our work contributes to the three main areas mentioned above as follows. First, we have presented the application of metamorphic testing to a novel domain, the analysis of feature models. In contrast to most related works, our metamorphic relations are derived from the operators of the models (i.e. types of relationships and constraints) rather than from the properties of the application domain in which they are used. Also, we have applied metamorphic testing in a slightly different way to the showed in related studies. In particular, we have used the metamorphic relations to compute the output of follow-up test cases instead of simply comparing the results of different tests. Starting from a trivial test case, we generate increasingly larger and more complex test data by making sure that the metamorphic relations are fulfilled at each step. This strategy allowed use to define the metamorphic relations for a single operation, *Products*, from which we derived the expected output of many of the other analyses on feature models. Second, we have presented a prototype tool for the automated generation of test data based on our metamorphic relations. In contrast to related works, we have evaluated our test data generator using hundred of automatically inserted mutants rather than manual mutation. We have also evaluated our approach with real faults found in the literature and current releases of several tools. We are not aware of any other study reporting the detection of real bugs using metamorphic testing. Finally, we have proposed a new integrated proposal combining our metamorphic approach and a black–box test suite showing experimental evidences of the gains obtained in terms of efficiency and efficacy.

8. Conclusions and Future Work

In this article, we presented a set of metamorphic relations on feature models and an automated test data generator based 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 operations 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 in detecting 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. Another two faults were detected in SPLOT, an online feature model analyzer actively used by the community. We also showed how our generator outperforms a related manual suite for the analysis of feature models. Finally, we explained how our approach can be refined by using a set of initial test cases that guide the generation of test data improving the detection of faults. Our results show that the application of metamorphic testing in the domain of automated analysis of feature models is efficient and effective in detecting most faults in a few seconds without the need for a human oracle. To the best of our knowledge, this is the first automated approach for functional testing on the analyses of feature models.

From a metamorphic testing point of view, our work shows that the definition of fairly simple metamorphic relations may lead to important fault detection rates at an affordable effort. We also show a novel application of metamorphic testing in which metamorphic relations are used to compute the output of follow-up test cases instead of comparing the output of different tests. This could certainly encourage researchers to explore new applications of metamorphic testing in similar domains in which the oracle problem appear. In this context, we plan to work in the definition of some generic guidelines to define metamorphic relations in similar data structures like those of variability models and configurators.

Material

Our prototype tool, the mutants and test classes used in our evaluation are available at `http://www.lsi.us.es/~segura/files/material/ist-10/`.

Acknowledgments

We would 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.

We would also like to thank the anonymous reviewers of the article whose comments and suggestions helped us to improve the article substantially.

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).

References

[1] Ahead tool suite. `http://userweb.cs.utexas.edu/users/schwartz/ATS.html`. accessed June 2010.

[2] D. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines Conference*, volume 3714 of *Lecture Notes in Computer Sciences*, pages 7–20. Springer–Verlag, 2005.

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

[4] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. In *Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, volume 3520 of *Lecture Notes in Computer Sciences*, pages 491–503. Springer–Verlag, 2005.

[5] 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.

[6] S. Beydeda. Self-metamorphic-testing components. In *Computer Software and Applications Conference, Annual International*, pages 265–272, September 2006.

[7] BigLever. Biglever software gears. `http://www.biglever.com/`. accessed June 2010.

[8] 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.

[9] W.K. Chan, T.Y. Chen, and H. Lu. A metamorphic approach to integration testing of context-sensitive middleware-based applications. In *QSIC '05: Proceedings of the Fifth International Conference on Quality Software*, pages 241–249, Washington, DC, USA, 2005. IEEE Computer Society.

[10] W.K. Chan, S.C. Cheung, and K. Leung. Towards a metamorphic testing methodology for service-oriented software applications. In *QSIC '05: Proceedings of the Fifth International Conference on Quality Software*, pages 470–476, Washington, DC, USA, 2005. IEEE Computer Society.

[11] T. Chen, J. Ho, H. Liu, and X. Xie. An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC Bioinformatics*, 10(1), 2009.

[12] 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.

[13] 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.

[14] 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.

[15] T.Y. Chen, F. Kuo, Y. Liu, and A. Tang. Metamorphic testing and testing with special values. In *Proceedings of the 5th International Conference on Software Engineering, Artificial Intelligence, Networking and Paralell/Distributed Computing*, 2004.

[16] T.Y. Chen, T.H. Tse, and Z. Zhou. Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 191–195. ACM, 2002.

[17] T.Y. Chen, T.H. Tse, and Z. Zhou. Fault-based testing without the need of oracles. *Information and Software Technology*, 45(1):1–9, 2003.

[18] Choco. `http://www.emn.fr/z-info/choco-solver/`. accessed June 2010.

[19] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison–Wesley, August 2001.

[20] 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*, 2005.

[21] 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.

[22] Fama framework. `http://www.isa.us.es/fama/`. accessed May 2010.

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

[24] D. Fernandez-Amoros, R. Heradio, and J. Cerrada. Inferring information from feature diagrams to product line economic models. In *Proceedings of the Sofware Product Line Conference*, 2009.

[25] Feature model plugin. `http://gp.uwaterloo.ca/fmp/`. accessed June 2010.

[26] R. Gheyi, T. Massoni, and P. Borba. A theory for feature models in alloy. In *Proceedings of the ACM SIGSOFY First Alloy Workshop*, pages 71–80, Portland, United States, nov 2006.

[27] A. Gotlieb and B. Botella. Automated metamorphic testing. In *COMPSAC '03: Proceedings of the 27th Annual International Conference on Computer Software and Applications*, page 34, Washington, DC, USA, 2003. IEEE Computer Society.

[28] Graphviz. `http://www.graphviz.org/`. accessed June 2010.

[29] Draft IEEE Standard for software and system test documentation (Revision of IEEE 829-1998). Technical report, 2007.

[30] Jacop. `http://jacop.osolpro.com/`. accessed May 2010.

[31] Javabdd. `http://javabdd.sourceforge.net/`. accessed May 2010.

[32] 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.

[33] 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.

[34] M. Mannion and J. Camara. Theorem proving for product line model verification. In *Software Product-Family Engineering (PFE)*, volume 3014 of *Lecture Notes in Computer Science*, pages 211–224. Springer Berlin / Heidelberg, 2003.

[35] M. Mendonca, M. Branco, and D. Cowan. S.P.L.O.T.: Software Product Lines Online Tools. In *Companion to the 24th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009*, pages 761–762, Orlando, Florida, USA, October 2009. ACM.

[36] M. Mendonça, A. Wasowski, and K. Czarnecki. SAT–based analysis of feature models is easy. In *Proceedings of the Sofware Product Line Conference*, 2009.

[37] Moskitt feature modeler. `http://www.pros.upv.es/mfm`. accessed June 2010.

[38] C. Murphy and G. Kaiser. Metamorphic runtime checking of non-testable programs. Technical Report cucs-012-09, Dept. of Computer Science, Columbia University, 2009.

[39] C. Murphy, K. Shen, and G. Kaiser. Automatic system testing of programs without test oracles. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 189–200, New York, NY, USA, 2009. ACM.

[40] C. Murphy, K. Shen, and G. Kaiser. Using JML runtime assertion checking to automate metamorphic testing in applications without test oracles. In *Conference on Software Testing, Verification, and Validation*, volume 0, pages 436–445, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[41] G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[42] Pure-systems. pure::variants. `http://www.pure-systems.com/`. accessed May 2010.

[43] Sat4j. `http://www.sat4j.org/`. accessed May 2010.

[44] P. Schobbens, J.C. Trigaux P. Heymans, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, Feb 2007.

[45] S. Segura, D. Benavides, and A. Ruiz-Cortés. FaMa Test Suite v1.2. Technical Report ISA-10-TR-01, ISA Research Group, 2010. Available at http://www.isa.us.es/.

[46] S. Segura, D. Benavides, and A. Ruiz-Cortés. Functional testing of feature model analysis tools: A test suite. *IET Software*, 2010. In press.

[47] 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.

[48] S. Segura, R.M. Hierons, D. Benavides, and A. Ruiz-Cortés. Mutation testing on an object-oriented framework: An experience report. *Information and Software Technology Special Issue on Mutation Testing*, 2010. In press.

[49] S. Segura, R.M. Hierons, D. Benavides, and A. Ruiz-Cortés. Mutation testing on an object-oriented framework: An experience report. Technical Report ISA-10-TR-02, ISA Research Group, June 2010. Available at http://www.isa.us.es/.

[50] 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.

[51] S.P.L.O.T.: Software Product Lines Online Tools. `http://www.splot-research.org/`. accessed May 2010.

[52] 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.

[53] Pablo Trinidad and Antonio Ruiz Cortés. Abductive reasoning and automated analysis of feature models: How are they connected? In *Third International Workshop on Variability Modelling of Software-Intensive Systems. Proceedings*, pages 145–153, 2009.

[54] P. van den Broek and I. Galvao. Analysis of feature models using generalised feature trees. In *Third International Workshop on Variability Modelling of Software-intensive Systems*, number 29 in ICB-Research Report, pages 29–35, Essen, Germany, January 2009. Universität Duisburg-Essen.

[55] Tijs van der Storm. Generic feature-based software composition. In *Software Composition*, volume 4829 of *LNCS*, pages 66–80. Springer, 2007.

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

[57] H. Wang, Y.F. Li, J. un, H. Zhang, and J. Pan. Verifying Feature Models using OWL. *Journal of Web Semantics*, 5:117–129, June 2007.

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

[59] J. White, D. Schmidt, D. Benavides P. Trinidad, and A. Ruiz-Cortes. Automated diagnosis of product-line configuration errors in feature models. In *Proceedings of the Sofware Product Line Conference*, 2008.

[60] X. Xie, W.E. Wong, T.Y. Chen, and B. Xu. Spectrum-Based Fault Localization Without Test Oracles. Technical report, Technical Report, UTDCS-7-10, Department of Computer Science, University of Texas at Dallas, 2010.

[61] W. Zhang, H. Mei, and H. Zhao. Feature-driven requirement dependency analysis and high-level software design. *Requirements Engineering*, 11(3):205–220, June 2006.

[62] 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.