Functional Testing of Feature Model Analysis Tools: A Test Suite

Sergio Segura, David Benavides and Antonio Ruiz-Cortés
Department of Computer Languages and Systems
Av Reina Mercedes S/N, 41012 Seville, Spain
{sergiosegura, benavides, aruiz}@us.es

Abstract: A Feature Model (FM) is a compact representation of all the products of a software product line. Automated analysis of FMs is rapidly gaining importance: new operations of analysis have been proposed, new tools have been developed to support those operations and different logical paradigms and algorithms have been proposed to perform them. Implementing operations is a complex task that easily leads to errors in analysis solutions. In this context, the lack of specific testing mechanisms is becoming a major obstacle hindering the development of tools and affecting their quality and reliability. In this article, we present FaMa Test Suite, a set of implementation–independent test cases to validate the functionality of FM analysis tools. This is an efficient and handy mechanism to assist in the development of tools, detecting faults and improving their quality. In order to show the effectiveness of our proposal, we evaluated the suite using mutation testing as well as real faults and tools. Our results are promising and directly applicable in the testing of analysis solutions. We intend this work to be a first step toward the development of a widely accepted test suite to support functional testing in the community of automated analysis of feature models.

Key Words: Test suite, functional testing, feature models, automated analysis, software product lines

1 Introduction

Software Product Line (SPL) engineering is an approach to develop families of related systems based on the usage of reusable assets as a means to improve software quality while reducing production costs and time–to–market [1]. Products in software product lines are specified in terms of features. A *feature* is defined as an increment in product functionality [2]. Key to SPLs is to capture commonalities (i.e. common features) and variabilities (i.e. variant features) of the systems that belong to the product line. To this aim, feature models are commonly used. A *feature model* [3, 4] is a compact representation of all the products of a product line in terms of features and relationships among them (see Figure 1).

The automated analysis of feature models deals with the automated 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 up to 30 different analysis operations on feature models have been reported in the literature [5, 6]. Analysis solutions can be categorized in those using propositional logic [7, 8, 9, 10, 11, 12, 13], constraint programming [14, 15, 16], description logic [17, 18] and ad-hoc algorithms and techniques [19, 20, 21]. There are also a number of tools supporting these analysis capabilities such as *AHEAD Tool Suite* [22], *FaMa Framework* [23], *Feature Model Plug-in* [24] and *pure::variants* [25].

The implementation of analysis operations is a hard task involving complex data structures and algorithms. This makes the development of analysis tools far from trivial and easily leads to errors increasing development time and reducing their reliability. Gaining confidence in the absence of defects in these tools is essential since the information extracted from feature models is used to support decisions all along the SPL development process [2]. However, 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. Hence, it is known that software testing accounts for about 50% of the total cost of software development [26].

Software testing intends to reveal faults in the software under test [27, 28]. This is mainly done by checking the behaviour of the program with set of input-output combinations (i.e. test cases). The main challenge when testing is to find a balance between the number of test cases and their effectiveness [28]. Hence, trying to be exhaustive when testing software tools may easily increase the number of test cases to an unmanageable level. Similarly, using a reduced number of input-output combinations may result in a weak effectiveness.

In this article, we present a set of implementation–independent test cases to validate the functionality of feature model analysis tools. Through the implementation of our test cases, faults can be rapidly detected assisting in the development of feature model analysis tools and improving their reliability and quality. For its design and evaluation, we used popular techniques from the software testing community to assist us on the creation of a representative set of input-output combinations. These test cases can be used either in isolation or as a suitable complement for further testing methods such as white–box testing techniques [27, 29] or automated test data generators [30]. As suggested by the testing literature, each test case was designed to reveal a single type of

fault. This allows users to identify clearly the source of a fault once it has been detected. Briefly, we next describe the main characteristics of our suite:

- **Operations tested.** Current version of our suite, called FaMa Test Suite, addresses 7 out of 30 analysis operations on feature models identified in the literature [5]. These were selected for their extended use in the community of automated analysis and their heterogeneous nature.
- **Testing techniques.** Four black-box testing techniques were used to design test cases, namely: equivalence partitioning, boundary-value analysis, pairwise testing and error guessing. A preliminary evaluation of the testing techniques to be used in our approach was presented in [31].
- **Test cases.** The suite is composed by 192 test cases. Each test case is designed in terms of the inputs (i.e. feature models and some other parameters) and expected outputs of the analysis operations under test.
- **Adequacy.** We evaluated the effectiveness of our suite using mutation testing and real faults as follows. Firstly, we generated hundreds of faulty versions (so-called mutants) of three open source analysis tools integrated into the FaMa framework. Then, we executed our suite against those faulty tools and check how many faults were detected by our test cases. As a result, the suite identified 96.1% of the faults showing the feasibility of our proposal. We then refined our suite until obtaining a score of 100%. Our refined suite also showed to be effective in detecting motivating faults found in the literature and in a recent release of the FaMa framework.

The remainder of the article is structured as follows: Section 2 presents feature models, their analyses and black-box testing techniques in a nutshell. A detailed description of how we designed our test cases is presented in Section 3. Section 4 describes the adequacy evaluation and refinement of the suite. A summary of the refined suite and a brief discussion is presented in Section 5. Finally, we summarize our main conclusions and describe our future work in Section 6.

2 Preliminaries

This section provides an overview of feature models, their analysis and the black–box testing techniques used in our approach.

2.1 Feature models

A feature model [3, 4] represents all the products of an SPL in a single model in terms of features and relationships among them. It is organized hierarchically and is graphically depicted as a *feature diagram*. Figure 1 shows a simplified example of a feature model representing an e-commerce SPL. The model illustrates how features are used to specify the commonalities and variabilities of the on-line shopping systems that belong to the product line. The root feature (i.e. E-Shop) identifies the SPL. The relationships between a parent feature and its child features can be mainly divided into:

Mandatory. A child feature has a mandatory relationship with its parent when the child 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. A child feature has an optional relationship with its parent when the child can be optionally included in all products in which its parent feature appears. For instance, *banners* is defined as an optional feature.

Alternative. A set of child features have an alternative relationship with their parent when only one feature of the children can be selected when its parent feature is part of the product. In our SPL, a shopping system may implement *high* or *medium* security policy but not both in the same product.

Or-Relation. A set of child features have an or-relationship with their parent when one or more of them can be included in the products in which its parent feature appears. A shopping system

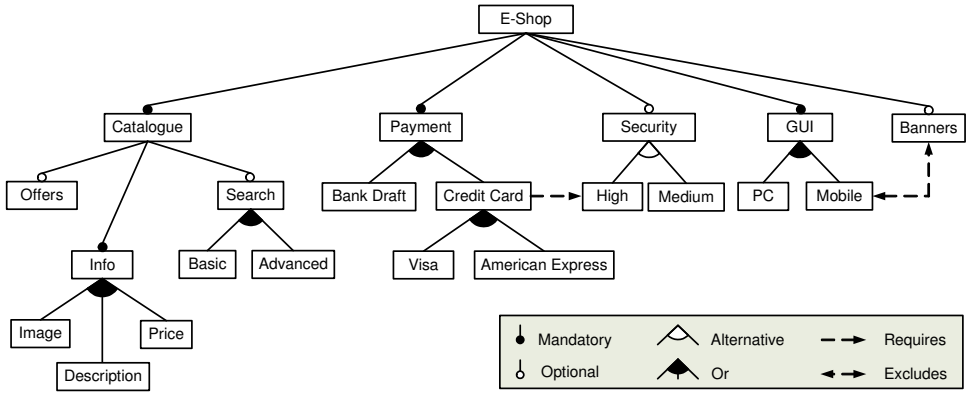


Figure 1: A sample feature model

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 cannot include support for *banners*.

Feature models were first introduced as a part of the FODA (Feature-Oriented Domain Analysis) method back in 1990 [4]. 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. We refer the reader to [6] for a detailed survey on the different feature modelling languages.

2.2 Automated analyses of feature models

The automated analysis of feature models deals with the computer–aided extraction of information from feature models [5]. From the information obtained, marketing strategies and technical decisions can be derived [32]. The analysis of a feature model is generally performed in two steps: *i*) First, the model is translated into a specific logic representation such as a satisfiability problem [7, 8, 9, 10, 11, 12, 13], a constraint satisfaction problem [14, 15, 16] or a knowledge base using description logic [17, 18], ii) Then, off-the-shelf solvers are used to automatically perform a variety of operations on the logic representation of the model. Catalogues with up to 30 different analysis operations on feature models have been reported in the literature [5, 6]. Following, we summarize some of the analysis operations we will refer 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 informing whether such feature model is void or not [6, 7, 8, 9, 10, 11, 12, 13, 14, 17, 18, 19, 20, 33]. A feature model is *void* if it represents no products.

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 [6, 7, 8, 9, 10, 14, 16, 18, 33]. As an example, P={*E-Shop, Catalogue, Info, Description, Security, Medium, GUI, PC*} is not a valid product of the product line represented by the model in Figure 1 because it does not include the mandatory feature *Payment*.

Obtaining all products. This operation takes a feature model as input and returns all the products represented by the model [7, 9, 10, 14, 19, 20, 33].

Calculating the number of products. This operation returns the number of products represented by a feature model [8, 10, 14, 19, 20, 21, 33]. The model in Figure 1 represents 2016 different products.

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 [14, 33]. 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 [14, 21, 33]. 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 [15]. In Figure 1, *Commonality(Banners)* = 25%.

Dead features detection. This operation takes a feature model as input and returns a set of dead features (if any) [2, 8, 11, 15, 33, 20, 13]. A *dead feature* is a feature that never appears in any of the products represented by the feature model [15]. These are caused by cross-tree constraints. Some common cases of dead feature are presented in Figure 4 (Section 3.2.1).

Some commercial and open source tools supporting the analysis of feature models are the *AHEAD Tool Suite* [22], *FaMa Framework* [23], *Feature Model Plug-in* [24] and *pure::variants* [25].

2.3 Black-box testing techniques

Software testing is performed by means of test cases. A *test case* is a set of test inputs and expected outputs developed to verify compliance with a specific requirement [29, 27]. Test cases may be grouped into so-called *test suites*. A variety of testing techniques has been reported to assist on the design of effective test cases, i.e. those that will find more faults with less effort and time [26, 27, 29]. In this article, we will focus on the so called *black-box* testing techniques. These techniques focus on checking whether a program does what it is supposed to do based on its specification [27]. No knowledge about the internal structure or implementation of the software under

test is assumed. We will refer to four of these techniques along the article. Briefly, these are:

Equivalence partitioning. This technique is used to reduce the number of test cases to be developed while still maintaining a reasonable test coverage (i.e. the degree to which the test cases verifies the test requirements) [27, 29]. In this technique, the input domain of the program is divided into partitions (also called *equivalence classes*) in which the program is expected to process the set of data input in a similar (i.e. equivalent) way. According to this testing approach, only one or a few test cases of each partition are needed to evaluate the behaviour of the program for the corresponding partition. Thus, selecting a subset of test cases from each partition is enough to test the program effectively while keeping a manageable number of test cases.

Boundary value analysis. This technique is used to guide the tester when selecting inputs from equivalence classes [27, 29]. According to this technique, programmers usually make mistakes with the inputs values located on the boundaries of the equivalence classes. This method guides the tester to select those inputs located on the "edges" of the equivalence partitions.

Pairwise testing (also called 2-wise testing). This is a combinatorial software testing method focusing on testing all possible discrete combinations of two input parameters [29, 34]. According to the hypothesis behind this technique, most common errors involve one input parameter. The next most common category of errors consists of those dependent on interactions between pairs of parameters and so on. Thus, the main goal of this technique is to address the second most common cases of errors (those involving two parameters) while keeping the number of test cases in a manageable level.

Error guessing. This is a software testing technique based on the ability of the tester to predict where faults are located according to its experience on the domain [29]. Using this technique, test cases are specifically designed to exercise typical error-prone points related to the type of system under test.

3 Test suite design

In this section, we describe how we designed the test cases that compose our suite. These mainly are input-output combinations specifically created to reveal failures in the implementations of analysis operations on feature models.

Creating test cases for every possible permutation of a program is impractical and very often impossible; there are simply too many input-output combinations [27]. Thus, as recalled by Pressman [28], the objective when testing is to *"design tests that have the highest likelihood of finding most errors with a minimum amount of time and effort"*. To assist us in the process, we evaluated a number of techniques reported in the literature [27, 29]. We focused on black-box techniques since we want our test cases to rely on the specification of the analysis operations rather than on specific implementations. In particular, we found the four techniques described in Section 2.3 to be effective and generic enough to be applicable to our domain.

For the design of the suite we followed four steps, namely: *i*) identification of the inputs and outputs of the analysis operations, *ii*) selection of representative instances of each type of input, *iii*) combination of previous instances in those operations receiving more than one input parameter, and *iv*) test cases report. Following, we detail how we carried out these steps.

3.1 Identification of inputs and outputs

The current version of our suite addresses 7 out of 30 analysis operations on feature models identified in the literature [5] (detailed in Section 2.2). We selected these operations for its extended use in the community of automated analysis and their heterogeneous nature. In order to identify the type of the input/output parameters of the operations and avoid misunderstandings when interpreting their semantics, we used the formal definition of the operations proposed by Benavides [32]. Table 1 summarizes the operations in terms of their inputs and outputs. For the sake of simplicity, we assign an identificator to each operation to refer them along the article. As illustrated, inputs are composed of feature models, products and features. Outputs mainly comprise collections of products and features together with numeric and boolean values.

The feature modelling notation used in our suite corresponds to one showed in Section 2.1

ID operation	Operation	Inputs	Output
VoidFM	Void FM	FM	Boolean value
ValidProduct	Valid product	FM, Product	Boolean value
Products	Products	FM	Collection of products
#Products	Number of products	FM	Numeric value
Variability	Variability	FM	Numeric value
Commonality	Commonality	FM, Feature	Numeric value
DeadFeatures	Dead features	FM	Collection of features

Table 1: Analysis operations addressed in the suite

(hereinafter referred as basic feature models). We selected this notation for its simplicity and extended use in current feature model analysis tools and literature.

3.2 Inputs selection

In this section, we explain how we selected the inputs to be used in our test cases. For each type of input (i.e. feature models, products and features), we next describe the techniques used and how we applied them.

3.2.1 Feature models

We found two testing techniques to be helpful for the selection of a suitable set of input feature models, namely: equivalence partitioning and error guessing. We applied them as follows:

Equivalence partitioning. The potential number of input feature models is limitless. To select a representative set of these, we propose dividing input feature models into equivalence classes according to the different types of relationships and constraints among features, i.e. mandatory, optional, or, alternative, requires and excludes. This is a natural partition intuitively used in most proposals when defining the mapping from a feature model to a specific logic paradigm (e.g. constraint satisfaction problem) [7, 9, 10, 13, 14, 15, 18, 32]. Therefore, according to this technique, if a feature model with a single mandatory relationship is correctly managed by an operation, we could assume that those with more than one mandatory relationship would also be processed successfully.

To keep equivalence classes in a manageable level, we propose dividing the input domain into three groups of partitions as follows:

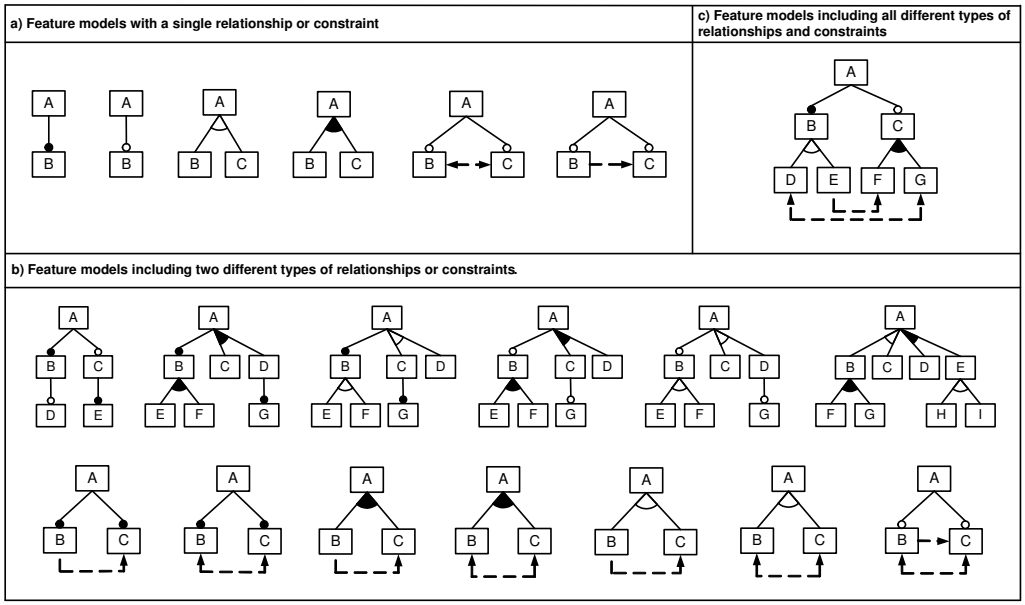


Figure 2: Equivalence partitions on feature models

1. *Feature models including a single type of relationship or constraint.* Inputs from these partitions would help us to reveal failures when processing isolated relationships and constraints. For basic feature models, 6 partitions are created: feature models including mandatory relationships, optional, or, alternative, requires and excludes. Figure 2 (a) depicts the inputs we selected from each equivalence class using this criterion. Note that feature models with requires and excludes constraints also include an additional relationship (e.g. optional) to make them syntactically correct, i.e. sharing a common parent feature.

2. *Feature models combining two different types of relationships or constraints.* We propose testing how analysis tools process feature model with multiple relationships by designing all the possible combinations of two of these, i.e. mandatory-optional, mandatory-or, etc. Following this criterion with basic feature models, we created a second group of 13 partitions. Figure 2 (b) presents the input models we took from each one of them. In general terms, feature relationships may appear in two main forms, child and sibling relationships (see Figure 3). According to our experience, inputs combining both types of relationships (highlighted in Figure 3) usually result in more complex constraints and therefore in more effective test cases. Thus, we selected inputs models from each partition randomly but making sure these included

both types of relationships, child and sibling relationships. For those input models including cross–tree constraints, we gave priority to simplicity and used sibling relationships exclusively.

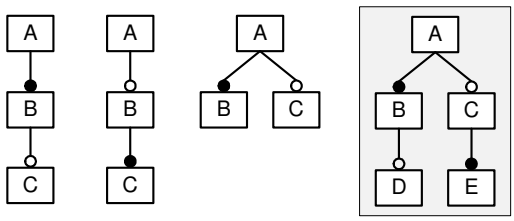


Figure 3: Some possible combinations of mandatory and optional relationships

3. *Feature models including three or more different types of relationships or constraints.* We finally propose creating a last partition including all those feature models not included in previous equivalence classes. Although this partition could be easily divided into smaller ones we did not find any evidence that justify such an increment in the number of test cases. Figure 2 (c) illustrates the random input feature model we took from this partition.

As a result of the application of this technique we got 20 feature models representing the whole input domain of basic feature models (see Figure 2). These were used as input in all the operations included in our suite with the exception of the operation *DeadFeatures* in which models derived from error–guessing were used.

Error guessing. Some common errors on feature models have been reported in the literature [15, 32]. As an example, Trinidad *et al.* [15] present some common problematic situations in which dead features appear (Figure 4). Following the guidelines of error guessing, we propose using these models as suitable inputs to check whether dead features are correctly detected by the tools under test (i.e. operation for the detection of dead features). This way, we kept the number of input models for this operation in a reasonable level while still having a fair confidence in the ability of our tests to reveal failures.

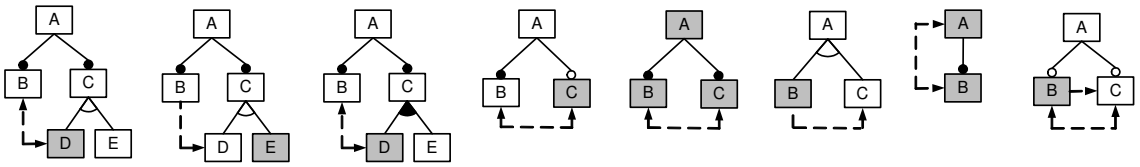


Figure 4: Input feature models with dead features (grey features are dead)

3.2.2 Products

Products are used in the operation *ValidProduct* to determine whether a given product (i.e. set of features) is included in those represented by a feature model. For the selection of these inputs, we propose applying equivalence partitioning and boundary analysis as follows.

Firstly, we propose dividing input products into two equivalent partitions: *valid* and *non-valid* products. For each of these equivalence classes, inputs should be treated in the same way by the program under test and should produce the same answer. Then, we propose using boundary analysis for the systematic selection of input products from each partition. In particular, we suggest quantifying products according to the number of features they include. Then, we propose selecting those products on the "edges" of the partitions.

Figure 5 depicts an example of how we applied equivalence partitioning and boundary analysis for the selection of input products. As input feature models, we used those presented in previous section created using equivalence partitioning (Figure 2). For each input feature model, four inputs products were selected, two valid and two non-valid, as follows:

- VP_{min} *: valid product with the minimum number of features.* This product could be helpful to reveal failures caused by spare of erroneous constraints when processing minimal solutions of the problem. For instance, a failure using $VP_{min} = \{A, B, D, E\}$ may suggest that any of the other features (i.e. C,F or G) are erroneously treated as core features. A *core* feature is a feature that is part of all the products [33].

- VP_{max}: *valid product with the maximum number of features.* This product would allow testers to detect overconstrained representations of the model using large valid combination of fea-

tures. As an example, a failure using $VP_{max} = \{A, B, C, E, G\}$ may suggest that some spare constraint forcing the selection of D or F is not being fulfilled. Note that VP_{min} would still be helpful to detect, for example, whether non core features are erroneously included in all products.

– NVP_{min}*: non-valid product with one less feature than* VP_{min}. This product would be helpful to reveal failures caused by omitted or insufficiently constrained representations of the models. In the example, $NVP_{min} = \{A, B, D\}$ could help us to check whether the parent feature of an alternative relationship (B) can be erroneously included in a product without including any of its child features (E or F).

– NVP_{max}*: non-valid product with one more feature than* VP_{max}. This product would help us to check broken constraint caused by the selection of too many features. For instance, $NVP_{max} = \{A, B, C, D, E, G\}$ may reveal failures derived from including in a product more than one alternative subfeature (C and D). Once again, we would still need NVP_{min} to make sure that underconstrained solutions are detected.

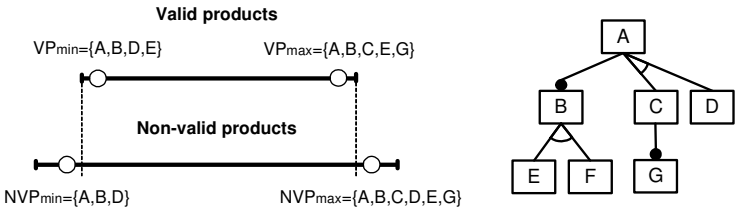


Figure 5: Input products selection using partition equivalence and boundary analysis

We identified two special causes making a product to be non-valid, namely: *i*) not including the root feature (e.g. product {B,D,E} for the model in Figure 5), and *ii*) including non-existent features (e.g. product {A,B,D,E,**H**} for the model in Figure 5). These causes are applicable to all products independently of the characteristics of the input feature model. Therefore, we did not consider these situations for products NVP_{min} and NVP_{max}. Instead of this, we checked these problems in two separated test cases to be as accurate as possible when informing about failures.

3.2.3 Features

Given a feature model, *Commonality* operation informs us about the percentage of products in which an input feature appears. For the selection of these input features, we propose applying equivalence partitioning and boundary analysis as follows.

To apply equivalence partitioning we suggest focusing on the result space of the commonality operation. More specifically, we propose considering a single output partition: from 0% to 100% of commonality. Then, we propose applying boundary analysis and selecting those input features returning a value situated on the edges of the output partition. Figure 6 depicts an example of our proposal. For each input feature model, two input features are used, one with minimum commonality (G=40%) and another one with maximum commonality (C=80%). We intentionally exclude the root feature whose commonality is trivial (100%). We also included an additional test case to check the behaviour of the operation when receiving a non-existent feature as input (e.g. feature H for model of Figure 6).

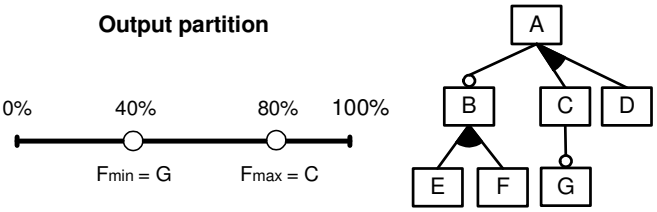


Figure 6: Input features selection using partition equivalence and boundary analysis

3.3 Inputs combination

Combination strategies define ways to combine input values in those programs receiving more than one input parameter [34]. This is the case of two of the operations included in our suite: *ValidProduct* (it receives a feature model and a product) and *Commonality* (it receives a feature model and a feature). To create effective test cases for these operations, we used a pairwise combination strategy (see Section 2.3). That is, we created a test case for each possible combination of the two input parameters. Hence, our suite satisfies 2-wise coverage being 2 the maximum number of input received by the operations included on it.

As an example, consider the operation *ValidProduct* which receives two inputs: a feature model and a product. In previous section, we studied these inputs in isolation and selected representative values for them resulting in 20 feature models (see Figure 2) and 4 products (i.e. VP_{min}, VP_{max}, NVP_{min} and NVP_{max}). Using pairwise testing, we created a test case for each possible combination of them (i.e. 20*4 potential test cases). Note that some combination were not applicable (e.g. feature models without non–valid products) reducing the number of test cases for this operation to 60.

3.4 Test cases report

To conclude the design of our suite, we organized the selected inputs and their expected outputs into test cases; 180 in total. For their specification, we followed the guidelines of the IEEE Standard for Software Testing Documentation [35]. As an example, Table 2 depicts three of the test cases included in the suite. For each test case, an ID, description, inputs, expected outputs and intercase dependencies (if any) are presented. Intercase dependencies refer to the identifiers of test cases that must be executed prior to a given test case. As an example, test case *P-9* tries to reveals failures when obtaining the products of feature models including mandatory and alternative relationships. Note that test cases *P-1* (test of mandatory relationship in isolation) and *P-4* (test of alternative relationship in isolation) should be executed beforehand. Test case *C-28* exercises the interaction between requires and excludes constraints when calculating commonality. Finally, test case *VP-37* is designed to reveal failures when checking whether an input product is included in those represented by a feature model including or- and alternative relationships. For a complete list of the test cases included in the suite we refer the reader to [36].

4 Test suite evaluation and refinement

In this section, we first detail the results obtained when using mutation testing to evaluate and refine our suite. Then, we show the efficacy of our tests in detecting some real faults found in the literature and a recent release of the FaMa Framework.

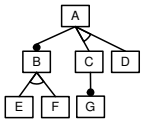
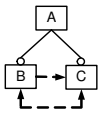
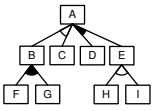
ID	Description	Input	Expected Output	Deps
P-9	Check whether the interaction between mandatory and alternative relationships is correctly processed.		{A,B,D,F}, {A,B,D,E}, {A,B,C,F,G}, {A,B,C,E,G}	P-1 P-4
C-28	Check whether the interaction between "requires" and "excludes" constraints is correctly processed. Input feature has minimum commonality.	 Feature=B	0%	C-2 C-5 C-6 C-7
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 2: Three of the test cases included in the suite

4.1 Evaluation using mutation testing

In order to measure the effectiveness of our proposal, we evaluated the ability of our test suite 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 [37] 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 (e.g. `i++ -> i--`). 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*. Detection of equivalent mutants is an undecidable problem in general. 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 analysis components (so-called reasoners) for the automated analysis of feature models. It is integrated into the feature modelling tool MOSKitt [38] and it is currently being integrated into the commercial tool pure::variants[1] [25] . Also, our familiarity with the tool made it feasible to use it for the mutations. In particular, we selected three of the reasoners integrated into the framework, namely: Sat4jReasoner v0.9.2 (using propositional logic by means of Sat4j solver [39]), JavaBDDReasoner v0.9.2 (using binary decision diagrams by means of JavaBDD solver [40]) and JaCoPReasoner v0.8.3 (using constraint programming by means of JaCoP solver [41]). 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.

For the mutations, we selected the key classes from each reasoner extending the framework and implementing the analysis capabilities. Table 3 shows size statistics of the subject programs including the number of classes selected from each reasoner, the total number of lines of code (LoC[2]) and the average number of methods and attributes per class. The size of the 28 subject classes included in our study ranged between 35 and 220 LoC.

Reasoner	**LoC**	**Classes**	**Av Methods**	**Av Attributes**
Sat4jReasoner	743	9	6.5	1.8
JavaBDDReasoner	625	9	6.3	2.1
JaCoPReasoner	791	10	6.3	2.3
Total	**2159**	**28**	**6.4**	**2.1**

Table 3: Size statistics of the three subject reasoners

To automate the mutation process, we used MuClipse Eclipse plug-in v1.3 [42]. MuClipse is a Java visual tool for object–oriented mutation testing. It supports a wide variety of operators and can be used for both generating mutants and executing them in separated steps. For the evaluation of the suite using MuClipse, we followed three steps, namely:

[1] In the context of the DiVA European project (http://www.ict-diva.eu/)
[2] LoC is any line within the Java code that is not blank or a comment.

1. *Reasoners testing.* Prior to their analysis, we made sure the original reasoners passed all the test cases in our suite.

2. *Mutants generation.* We applied all the traditional mutation operators available in MuClipse, a total of 15. These mainly mutate common programming languages features such as arithmetic (e.g. `++`,`--`) and relational operators (e.g. `==`,`!=`). Specific mutation operators for object–oriented code were discarded to keep the number of mutants manageable. Once generated, we manually discarded those mutants affection portions of the code not related to the analysis of feature models and therefore not addressed by our test suite (e.g. exception handling). For details about the mutation operators used in our evaluation we refer the reader to [43].

3. *Mutants execution.* For each mutant, we ran our test cases using JUnit [44] and tried to kill it. We may remark 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, i.e. some mutants were executed more than once. As a result, the number of executed mutants was higher than the number of generated mutants. Equivalent mutants were manually identified and discarded after each execution.

4.1.2 Analysis of results

Table 4 shows information about the number of generated mutants. Out of the 749 generated mutants, 101 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 other aspects of the program not related to the analysis of feature models and therefore not considered in our test suite. These were mainly related to the computation of statistics (e.g. execution time) and exception handling.

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

Table 4: Mutants generation results

Table 5 depicts the results obtained when using our test suite to kill the mutants in the FaMa reasoners. For each operation and reasoner, the total number of mutants executed, number of alive mutants and mutation score are presented. The detection of dead features was tested only in JaCoPReasoner since this was the only reasoner implementing it. As illustrated, the operations *VoidFM* and *ValidProduct* produced the lowest scores and higher number of alive mutants. We found that mutants on these operations required input models to have a very specific pattern in order to be killed and therefore were harder to detect than mutants in the rest of operations. Average mutation scores in the three reasoners ranged between 94.4% and 100%. In total, our test suite was able to kill 1390 (96.1%) out of the 1445 mutants executed showing the effectiveness of the suite.

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 5: Mutants execution results

4.1.3 Refinement

As shown in previous sections, mutation testing is an effective means to measure the effectiveness of a test suite. However, information provided by mutation testing can also be used to guide the creation of new test cases that kill the remaining alive mutants and strengthen the final test suite [42]. Following this approach, we designed a number of test cases to kill remaining undetected mutants until obtaining a score of 100% in the three FaMa reasoners. A total of 27 new test cases were created and executed. For instance, alive mutants guided us to the creation of a couple test cases to ensure that alternative relationships are not processed as or-relationships and vice–versa (e.g. test cases *VM-21* and *VM-22* in [36]). Out of the 27 test cases created, we selected those test cases that showed to be effective in killing mutants in at least two of the three subject reasoners

and added them to our suite. As a result, 10 test cases were added to the initial test suite increasing the number of these until 190.

4.2 Evaluation using real faults

For a further evaluation of our approach, we checked the effectiveness of our tool in detecting real faults. In particular, we first studied a motivating fault found in the literature. Then, we used our test suite to test the release 1.0 alpha of the FaMa framework, detecting one defect. These results are next reported.

4.2.1 Motivating fault found in the literature

Consider the work of Batory in SPLC'05 [7], one of the seminal papers in the community of automated analysis of feature models. The paper included a bug (later fixed[3]) in the mapping of a feature model to a propositional formula. We implemented this wrong mapping into a mock reasoner for FaMa and checked the effectiveness of our approach in detecting the fault.

Figure 7 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 and excluding the parent feature (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 3,584 (instead of the actual 2,016). 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.

We implemented our test cases using JUnit and tested our faulty tool. The fault was detected by our test suite in the operations *VoidFM*, *Product*, *#Products*, *Variability* and *Commonality* remaining latent in the operations *ValidProduct* and *DeadFeatures*. These two operations required a very specific pattern to reveal the fault not included in the inputs of our test suite. This gives

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

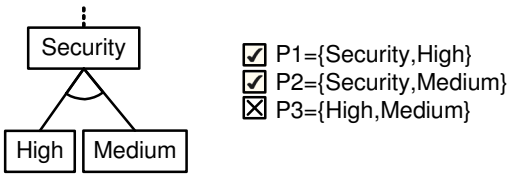


Figure 7: Wrong set of products obtained with the faulty reasoner

an idea of the complexity of testing in this domain. We found this fault sufficiently motivating to extend our suite with test cases that detect it in all the operations. Thus, we designed two new test cases to detect the fault in the operation *ValidProduct* and *DeadFeatures* and added them to our test suite resulting in a total of 192 test cases.

4.2.2 FaMa v1.0 alpha

Finally, we evaluated our tool by trying to detect faults in a recent release of the FaMa Framework, *FaMa v1.0 alpha*. We executed the 192 test cases of our refined test suite. Tests revealed one defect. The 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 Test suite summary and discussion

Table 6 summarizes the general aspects of the refined test suite using the common terms of the IEEE Standard for Software Testing Documentation [35]. For each operation, the number of test cases and the testing techniques used are presented. Global inputs constraints specify constraints that must be true for every input in the set of associated test cases. For the sake of simplicity, two main input constraints were imposed, namely: *i*) input feature models must be syntactically correct (e.g. checking models for conformance to a metamodel), and *ii*) all input parameters of the analysis operations must be provided. An operation is said to pass the test (so-called pass criteria) when all the test cases associated to that operation are successful.

Trying to be exhaustive when testing feature model analyses tools can easily increase the number of test cases to an unmanageable level. To keep a reasonable balance between number

of test cases and test coverage, we kept in mind a number of generic recommendations from the testing literature, namely: *i*) we gave priority to those decisions reducing the number of test cases, *ii*) we avoided redundancies by designing each test case to reveal a single type of fault, and *iii*) we designed simple test cases whose output could be worked out manually to avoid test themselves to become error-prone.

We remark that the potential users of the suite are every tool supporting the analysis of feature model. This can be automated by simply implementing the test cases in the desired platform and executing them. A complete list of the test cases that compose the suite is reported in [36]. To facilitate its implementation, input models used in the test cases are also available in XML format in the FaMa Tool Suite Web site[4].

Test suite identifier: FaMa Test Suite v1.2		
Operations tested	**Test cases**	**Techniques used**
Void FM	24	EP, PT
Valid product	63	EP, PT, BVA
Products	21	EP, PT
Number of products	21	EP, PT
Variability	21	EP, PT
Commonality	33	EP, PT, BVA
Dead features	9	EG
Total	**192**	
Global input constraints: - Input FMs must be syntactically correct - All input parameters are required		
Operation pass criteria: - Pass 100% of associated test cases		

Table 6: General overview of the FaMa Test Suite (EP: Equivalence Partitioning, PT: Pairwise Testing, BVA: Boundary-Value Analysis, EG: Error Guessing)

6 Conclusions and future work

In this article, we present a set of implementation–independent test cases to validate the functionality of tools supporting the analysis of feature models. Through the implementation of our test cases, faults can be rapidly detected assisting in the development of feature model analysis tools and improving their reliability and quality. These can be used either in isolation or as a suitable complement for further testing methods such as white–box testing techniques or automated test

[4] http://www.isa.us.es/fama/?FaMa_Test_Suite

data generators. For its design, we used popular techniques from the software testing community to assist us on the creation of a representative set of input–output combinations. To evaluate its effectiveness, we applied mutation testing on three open source feature model analysis tools integrated into the FaMa framework. These tools use different underlying paradigms and were coded by different developers what provides the necessary heterogeneity for the evaluation. We initially obtained an average mutation score of 96.1% and refined our suite progressively until getting 100%. Once refined, our suite also showed to be effective in detecting real faults found in the literature and in a recent release of FaMa. Both, the test suite documentation and the inputs models used in the test cases are ready–to–use and available at the Web Site of the FaMa Tool Suite. We intend this article to be a first effort toward the development of a widely accepted test suite to support functional testing in the community of automated analysis of feature models.

Several challenges remain for our future work in two main directions, namely:

- We intend to extend our suite with new operations and testing techniques. We also plan to evaluate our suite with other tools and in development scenarios using FaMa and report our experiences to the community.

- We also plan to explore the benefits obtained when combining our suite with other automated testing methods. In a previous work [30], we presented an automated test data generator for the analysis of feature models with promising results. It generates random follow–up test cases based on the relations between inputs feature models and theirs expected outputs (so-called metamorphic testing). However, it is known that metamorphic testing produce better results when combined with other test case selection strategies that generate the initial set of test cases. We intend to use our suite to guide the generation of test cases in our automated test data generator and study the gains in efficiency and efficacy.

Material

The tools, mutants, and test cases used in our evaluation are available at `http://www.lsi.us.es/~segura/files/material/fts_1_2/`

Acknowledgments

We would like to thank the reviewers of the article as well as Don Batory and Robert M. Hierons whose comments and suggestions helped us to improve the article substantially. We also thank Jose Galindo for his technical support during the evaluation of our approach.

This work has been partially supported by the European Commission (FEDER), Spanish Government under CICYT projects SETI (TIN2009-07366) and Web-Factories (TIN2006-00472) and the Andalusian Government project ISABEL (TIC-2533).

References

1. Clements P, Northrop L. Software Product Lines: Practices and Patterns. SEI Series in Software Engineering. Addison–Wesley; 2001.
2. Batory D, Benavides D, Ruiz-Cortés A. Automated Analysis of Feature Models: Challenges Ahead. Communications of the ACM. 2006;December:45–47.
3. Czarnecki K, Eisenecker UW. Generative Programming: Methods, Techniques, and Applications. Addison–Wesley; may 2000. ISBN 0–201–30977–7.
4. Kang K, Cohen S, Hess J, Novak W, Peterson S. Feature–Oriented Domain Analysis (FODA) Feasibility Study. SEI; 1990. CMU/SEI-90-TR-21.
5. Benavides D, Segura S, Ruiz-Cortés A. Automated Analysis of Feature Models 20 Years Later: A Literature Review. Information Systems. 2010;In press. Available from: `Availableathttp://dx.doi.org/10.1016/j.jss.2010.02.017`.
6. Schobbens P, P Heymans JCT, Bontemps Y. Generic semantics of feature diagrams. Computer Networks. 2007 Feb;51(2):456–479.
7. Batory D. Feature Models, Grammars, and Propositional Formulas. In: Software Product Lines Conference, LNCS 3714; 2005. p. 7–20.
8. Czarnecki K, Kim P. Cardinality-Based Feature Modeling and Constraints: A Progress Report. In: Proceedings of the International Workshop on Software Factories At OOPSLA 2005; 2005. .
9. Gheyi R, Massoni T, Borba P. A Theory for Feature Models in Alloy. In: Proceedings of the ACM SIGSOFY First Alloy Workshop. Portland, United States; 2006. p. 71–80. Available from: `http://alloy.mit.edu/workshop/programme.html`.
10. Mannion M, Camara J. Theorem Proving for Product Line Model Verification. In: Software Product-Family Engineering (PFE). vol. 3014 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg; 2003. p. 211–224. Available from: `http://www.springerlink.com/content/m0k40djlmmxx8vtp/`.
11. Mendonça M, Wasowski A, Czarnecki K. SAT–based analysis of feature models is easy. In: Proceedings of the Sofware Product Line Conference; 2009. .

12. van der Storm T. Generic Feature-Based Software Composition. In: Software Composition. vol. 4829 of LNCS. Springer; 2007. p. 66–80.
13. Zhang W, Mei H, Zhao H. Feature-driven requirement dependency analysis and high-level software design. Requirements Engineering. 2006 June;11(3):205–220. Available from: http://www.springerlink.com/content/v1648q73m788q71x/.
14. Benavides D, Ruiz-Cortés A, Trinidad P. Automated Reasoning on Feature Models. LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005. 2005;3520:491–503.
15. Trinidad P, Benavides D, Durán A, Ruiz-Cortés A, Toro M. Automated Error Analysis for the Agilization of Feature Modeling. Journal of Systems and Software. 2008;81(6):883–896.
16. White J, Schmidt D, Trinidad DBP, Ruiz-Cortés. Automated Diagnosis of Product-line Configuration Errors in Feature Models. In: Proceedings of the 12th Sofware Product Line Conference (SPLC'08). Limerick, Ireland; 2008. .
17. Fan S, Zhang N. Feature Model Based on Description Logics. In: Knowledge-Based Intelligent Information and Engineering Systems; 2006. Available from: http://dx.doi.org/10.1007/11893004_145.
18. Wang H, Li YF, un J, Zhang H, Pan J. Verifying Feature Models using OWL. Journal of Web Semantics. 2007 June;5:117–129. Available from: http://dx.doi.org/10.1016/j.websem.2006.11.006.
19. van Deursen A, Klint P. Domain–Specific Language Design Requires Feature Descriptions. Journal of Computing and Information Technology. 2002;10(1):1–17.
20. van den Broek P, Galvao I. Analysis of Feature Models using Generalised Feature Trees. In: Third International Workshop on Variability Modelling of Software-intensive Systems. No. 29 in ICB-Research Report. Essen, Germany: Universität Duisburg-Essen; 2009. p. 29–35. Available from: http://www.vamos-workshop.net/proceedings/VaMoS_2009_Proceedings.pdf.
21. Fernandez-Amoros D, Heradio R, Cerrada J. Inferring Information from Feature Diagrams to Product Line Economic Models. In: Proceedings of the Sofware Product Line Conference; 2009. .
22. AHEAD Tool Suite. http://www.cs.utexas.edu/users/schwartz/ATS.html;. Accessed November 2009.
23. Benavides D, Trinidad P, Segura S, Ruiz-Cortés A. FaMa Framework. http://www.isa.us.es/fama/;.
24. Feature Modeling Plug-in. http://gp.uwaterloo.ca/fmp/;. Accessed November 2009.
25. pure::variants. http://www.pure-systems.com/;. Accessed November 2009.
26. Beizer B. Software testing techniques (2nd ed.). New York, NY, USA: Van Nostrand Reinhold Co.; 1990.
27. Myers GJ, Sandler C. The Art of Software Testing. John Wiley & Sons; 2004.
28. Pressman RS. Software Engineering: A Practitioner's Approach. 5th ed. McGrap-Hill; 2001.
29. Copeland L. A Practitioner's Guide to Software Test Design. Norwood, MA, USA: Artech House, Inc.; 2003.
30. Segura S, Hierons RM, Benavides D, Ruiz-Cortés A. Automated Test Data Generation on the Anal-

yses of Feature Models: A Metamorphic Testing Approach. In: International Conference on Software Testing, Verification and Validation. Paris, France: IEEE press; 2010. In press.

31. Segura S, Benavides D, Ruiz-Cortés A. Functional Testing of Feature Model Analysis Tools. A First Step. In: 5th Software Product Lines Testing Workshop (SPLiT 2008). SPLC'08. Limerick, Ireland; 2008. .
32. Benavides D. On the Automated Analyisis of Software Product Lines using Feature Models. A Framework for Developing Automated Tool Support. University of Seville; 2007.
33. Trinidad P, Ruiz-Cortés A. Abductive Reasoning and Automated Analysis of Feature Models: How are they connected? In: Third International Workshop on Variability Modelling of Software-Intensive Systems. Proceedings; 2009. p. 145–153. Available from: `http://www.vamos-workshop.net/proceedings/VaMoS_2009_Proceedings.pdf`.
34. Grindal M, Offutt J, Andler SF. Combination testing strategies: a survey. Software Testing, Verification and Reliability. 2005;15(3):167–199. Available from: `http://dx.doi.org/10.1002/stvr.319`.
35. Draft IEEE Standard for software and system test documentation (Revision of IEEE 829-1998); 2007. Available from: `http://ieeexplore.ieee.org/xpls/abs\_all.jsp?arnumber=4432350`.
36. Segura S, Benavides D, Ruiz-Cortés A. FaMa Test Suite v1.2. ISA Research Group; 2010. ISA-10-TR-01. Available at http://www.isa.us.es/.
37. DeMillo RA, Lipton RJ, Sayward FG. Hints on Test Data Selection: Help for the Practicing Programmer. IEEE Computer. 1978;11(4):34–41.
38. Cetina C, Fons J, Pelechano V. Moskitt Feature Modeler. `http://www.pros.upv.es/mfm`;. Accessed November 2009.
39. Sat4j. `http://www.sat4j.org/`;. Accessed November 2009.
40. JavaBDD. `http://javabdd.sourceforge.net/`;. Accessed November 2009.
41. JaCoP. `http://jacop.osolpro.com/`;. Accessed November 2009.
42. Smith BH, Williams L. On guiding the augmentation of an automated test suite via mutation analysis. Empirical Software Engineering. 2009;14(3):341–369.
43. Ma YS, Offutt J. Description of Method-level Mutation Operators for Java, `http://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf`; 2005. Accessed 2/10/2009.
44. JUnit. `http://www.junit.org/`;. Accessed November 2009.