Grammar-based Test Generation for Software Product Line Feature Models

Ebrahim Bagheri, Faezeh Ensan, Dragan Gasevic
Ryerson University, University of British Columbia, Athabasca University
bagheri@ryerson.ca, faezeh.ensan@sauder.ubc.ca, dgasevic@acm.org

Abstract

Product lines are often employed for the facilitation of software re-use, rapid application development and increase in productivity. Despite the numerous advantages of software product lines, the task of testing them is a cumbersome process due to the fact that the number of applications that need to be tested is exponential to the number of features represented in the product line. In this paper, we attempt to reduce the number of required tests for testing a software product line while at the same time preserving an acceptable fault coverage. For this purpose, we introduce eight coverage criteria based on the transformation of software product line feature models into formal context-free grammars. The theoretical foundation for the proposed coverage criteria is based on the development of equivalence partitions on the software product line configuration space and the use of boundary value analysis for test suite generation. We have performed experiments on several SPLOT feature models, the results of which show that the test suite generation strategies based on the proposed coverage criteria are effective in significantly reducing the number of required tests and at the same time maintaining a high fault coverage ratio.

1 Introduction

Large and complex domains are a potential venue for the development of many different software applications. These applications can share a lot of similarities due to the fact that they have been developed for the same target domain and also have differences based on the nature of the specific problem that they are trying to solve within the target domain. The concept of software product lines is amongst the widely used means for capturing and handling these inherent *commonalities* and *variabilities* of the applications of a target domain [9, 26]. Within the realm of software product lines, these similarities and differences are viewed in terms of the core competencies and functionalities, referred to as *features*, provided by each of the applications [21, 18]. Therefore, a software product line is a model of a domain formalizing the existence of and the interactions between the features.

The problem space of a software product line is often represented through *feature models*, which are tree-like structures whose nodes are the domain features and the edges are the interactions between the features [17]. Each feature model is an abstract representation of the possible applications of a target domain; therefore, it is possible to develop new applications from a feature model by simply selecting a set of most desirable features from the feature model. Since a feature model is a generic representation of all possible applications of a domain, the selection of a set of features from the feature model yields a specific application. This process is referred to as *feature model configuration* [12]. It is clear that the selection of different features from the feature model results in different feature model configurations and hence different software applications. For this reason, it is reasonable to say that a single feature model can be configured in different ways to form numerous applications. As a matter of fact, the number of possible configurations of a feature model increases exponentially with the size of the feature model [20].

This observation leads to the main concern with regards to testing software product lines and their products namely the time and effort required for testing all of the possible applications of a software product line. Outside the context of software product lines, testing often involves a single application that needs to be fully analyzed; however, the process of testing a software product line and ensuring that it is fault-free requires the comprehensive analysis of all of its potential products. Assuming that a feature model contains n features and that each application configured from that feature model takes $O(m)$ to be tested, a complete test of the applications derivable from that feature model would in the worst case take $O(2^n \times m)$, which is impractical both in terms of the required resources to generate all of the tests and also the time needed for performing the tests. This necessitates the need for developing test generation strategies that would create small but efficient test suites for testing large software product lines. As will be discussed in Section 6, these problems have already been recognized as important in the SPL community, but are yet to be fully explored.

In short, feature models are representatives of a magnitude of applications (aka products) that can be derived from the feature model through the configuration process [5]. Therefore, one can only be certain that a feature model is safe (fault-free) iff all of the possible applications of the feature model are comprehensively tested. Given that this requires significant resources (time and effort), in this paper we propose a set of coverage criteria in order to select a smaller set of applications from among all possible applications of a feature model. These selected applications will be then comprehensively tested instead. Throughout this paper, a *test suite* is a restricted collection of applications derived from a feature model based on certain coverage criteria; hence a test suite consists of several applications to be tested independently. We refer to each member of the test suite as a *test*. The members of a test suite, i.e. the tests, are the selected applications that are to be tested comprehensively instead of the whole feature model application space. Our goal is to create new coverage criteria that will allow us to generate small test suites, i.e., to select a small subset of the product line application space. The hypothesis is that the set of applications identified based on the coverage criteria are able to reduce the test space while maintaining a high fault coverage.

In this paper, we propose to view software product line feature models expressed in terms of context-free grammars represented in Extended Backus-Naur Form (EBNF) [11] and to extend the existing coverage criteria for EBNF by proposing eight new coverage criteria for feature models. These coverage criteria form the basis for the generation of smaller test suites, which are at the same time quite efficient in identifying faults in feature models. Our work is based on the concepts of *boundary value analysis* and *equivalence partitioning* [24] alongside the coverage criteria for generating efficient test suites for software product lines. More specifically, our work provides the following three main contributions:

1. We provide the means for viewing software product line feature models in terms of context-free (EBNF) production rules. This allows us to define eight main test coverage criteria for testing feature models – which is one of the first in the area of product lines;

2. Given the context-free grammar representation of feature models and the formal definition of the coverage criteria, different test suite generation strategies are proposed that would allow for the automatic development of test suites. These are fundamentally based on equivalence partitioning and boundary conditions;

3. Each of the proposed strategies are employed for generating test suites for nine SPLOT feature models. The purpose is to evaluate the fault coverage performance of the proposed test generation strategies.

The main distinguishing aspects of our work from the other related work are: 1) the main techniques in the area of product line testing focus on t-wise and combinatorial testing strategies [25, 30, 31]. However, our work centers around the definition of a set of semantically well defined coverage criteria, which are the basis for our test generation strategies. In our view, this is significant because the outcomes of the testing process based on each coverage criterion provides insight into the possible issues with the product line in light of the semantics of the coverage criterion used. Therefore, the tester would know how to trace the results back to their origin based on the purpose of the coverage criterion that was employed in that case. It should be noted here that traceability is from the test that revealed a fault to the coverage criteria that was the origin of that test (and not between the specific test and the feature causing the fault); 2) the introduction of the eight coverage criteria allows the product line tester to customize the test suites and test cases based on the specific circumstances that are present. This is different from the related work where the generated test suites for the product line cannot be controlled by the product line tester. Furthermore, the tester will not completely know what implications his changes (addition or removal of tests) would have on the coverage of the testing process in those approaches. However in our work, the tester is able to control the generated test suites based on the coverage criterion that he/she selects to use and would have knowledge about why each test was included; 3) we provide and share the implementation of all our test suite generation strategies available at `http://ebagheri.athabascau.ca/splt/splt.zip`. Furthermore, we point to the publicly available feature models that were used in our experiments for future replication and comparative studies.

The rest of the paper is organized as follows: the next section covers the preliminaries regarding the structure of feature models and how they are expressed using context-free grammars. Section 3 provides the theoretical basis of our approach. The coverage criteria for feature models and their corresponding test generation strategies are presented in Section 4. Section 5 provides the details of the evaluation of the test generation strategies; followed by related work in Section 6. The paper is then concluded in Section 7.

2 Preliminaries

2.1 Feature Models

Features are important distinguishing aspects, qualities, or characteristics of a family of systems [21, 17]. They are widely used for depicting the shared structure and behavior of a set of similar systems. To form a product family, all the various features of a set of similar/related systems are composed into a feature model. Feature models can be represented both formally and graphically; however, the graphical notation depicted through a tree structure is more favored due to its visual appeal and easier understanding. More specifically, graphical feature models are in the form of a tree whose

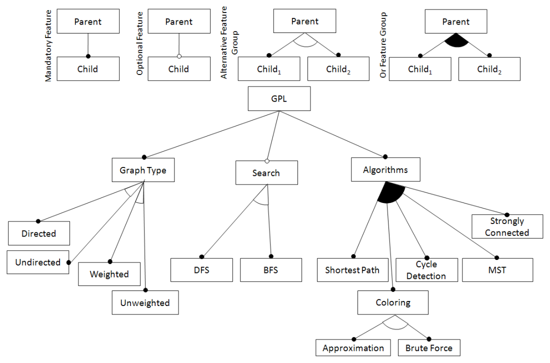

Figure 1: The GPL feature model.

root node represents a domain, and the other nodes and leafs illustrate the features.

In a feature model, features are hierarchically organized and can typically be classified as: *Mandatory*, *Optional*, *Alternative feature group*, and *Or feature group*. Figure 1 depicts the graphical notation of the feature relationships. The tree structure of feature models falls short at fully representing the complete set of mutual interdependencies of features; therefore, additional constraints are often added to feature models and are referred to as *Integrity Constraints (IC)*. The two most widely used integrity constraints are: 1) *Includes*, the presence of a given feature (set of features) requires the *existence* of another feature (set of features); and 2) *Excludes*, the presence of a given feature (set of features) requires the *elimination* of another feature (set of features).

The Graph Product Line (GPL) [22] depicted in Figure 1 is the classical sample feature model in the software product line community that covers the classical set of applications of graphs in the domain of Computer Science. As it can be seen, GPL consists of three main features: 1) `Graph Type`: features for defining the structural representation of a graph; 2) `Search`: traversal algorithms in the form of features that allow for the navigation of a graph; 3) `Algorithms`: other useful algorithms that manipulate or analyze a given graph. Clearly, not all possible configurations of the features of GPL produce valid graph programs. For instance, a configuration of GPL that checks if a graph is strongly connected cannot be implemented on an undirected graph structure. Such restrictions are expressed as integrity constraints. Some examples of such constraints are: `Cycle Detection` EXCLUDES `BFS`[1] and `Cycle Detection` INCLUDES `DFS`.

The integrity constraints and the structure of the feature model ensure that correct product configurations are derived from a feature model. For instance, the feature model in Figure 1 can be configured in 308 different ways; hence, having the potential to produce 308 domain-dependent applications. As mentioned earlier, a comprehensive strategy for testing feature models is to test all of the possible configurations. In other words, each possible configuration is a *test* for evaluating the feature model. Taking this approach for testing the graph product line feature model would require 308 different applications to be tested which is not practical given such a small feature model.

2.2 Feature Models in EBNF

Czarnecki and Eisenecker have argued that the grammar of feature models can be represented in EBNF [11]. In this paper, we employ the EBNF notation in order to represent a feature model in the form of a set of production rules. EBNF is a family of syntactical notations that are used for describing context-free grammars and are as such in the form of $V \rightarrow w$, where V is a single nonterminal symbol, and w is a string of terminals and/or nonterminals or empty productions.

It is easy to see that the hierarchical structure of feature models can be represented using context-free grammars denoted using the EBNF notation. Each parent feature can be the left-hand side of a production rule while its right-hand side will be its child features. For instance, the search feature in GPL can be represented in EBNF as `Search` $\rightarrow$ `t_DFS | t_BFS;`[2] or GPL root can be depicted as `GPL` $\rightarrow$ `GraphType, Algorithms | GraphType, Algorithms, Search;`

To be able to represent integrity constraints, additional production rules need to be inserted. For instance, to show that `Cycle Detection` excludes `BFS` and includes `DFS`, the related production rules would be rewritten as follows:

1. `GPL` $\rightarrow$ `GraphType, Algorithms`$_1$`, Search`$_1$ `|` `GraphType, Algorithms`$_2$`, Search`$_2$`;`
2. `Search`$_1$ $\rightarrow$ `t_DFS;`
3. `Search`$_2$ $\rightarrow$ `t_DFS | t_BFS;`
4. `Algorithms`$_1$ $\rightarrow$ `t_CycleDetection, Algorithms`$_2$`;`
5. `Algorithms`$_2$ $\rightarrow$ `Coloring, Temp`$_1$ `| Temp`$_1$`;`
6. `Temp`$_1$ $\rightarrow$ `t_MST, Temp`$_2$ `| Temp`$_2$`;`
7. `Temp`$_2$ $\rightarrow$ `t_StronglyConnected, Temp`$_3$ `| Temp`$_3$`;`
8. `Temp`$_3$ $\rightarrow$ `t_ShortestPath |` λ`;`
9. `Coloring` $\rightarrow$ `t_Approximation | t_BruteForce;`

where λ is an empty production.

As depicted above, additional production rules have been created, shown with subscript, to make sure that the EBNF products respect the two integrity constraints. In this example, the additional productions will ensure that `Cycle Detection` and `BFS` can never be seen together and also whenever `Cycle Detection` is produced that `DFS` is also generated as a requirement for it. We have developed the required software program that would automatically convert a feature model represented in the standard SXFM feature model format into EBNF production rules. The source code of all our work is available at `http://ebagheri.athabascau.ca/splt/splt.zip`.

It is noteworthy that the use of a context-free grammar-based representation is quite useful for our purpose because our focus is on test generation and these tests need to be valid applications derivable from the feature model; therefore, an EBNF-based representation will make sure that only valid tests are generated based on the available production rules. Furthermore, although as we will show later in the paper that the only operation required by our work is the words (strings) generation operation, still more complex operations such as finiteness of the grammar or word-grammar membership are *decidable* in context-free grammars (type-2 grammars). For this reason, such rewriting of feature models into context-free grammars is suitable for future extensions of our work, e.g., such a

[1] breadth-first and depth-first search

[2] We denote non-terminals with capitalization and terminals with t_.

grammar could allow one to check if some product is a valid configuration based on the feature model grammar representation.

In the following, we will show how the representation of feature models in EBNF can be used to develop several coverage criteria and test generation strategies.

3 Theoretical Basis

We base the development of our coverage criteria on the work of Ammann and Offutt on BNF coverage criteria [1]. These authors provide three fundamental test coverage criteria for BNF grammars as follows:

Definition 1 ***Derivation Coverage*** *[1] – The* test suite *must contain every possible string derivable from grammar G.*

In this definition and throughout this paper, a *test suite* is a collection of tests. Also, a *coverage criterion* is a rule or collection thereof that imposes the presence or absence of a specific aspect of the software artifact in a test suite. Therefore, Derivation Coverage is a coverage criterion which would require all of the possible strings produced by a BNF grammar to be tested. This is an *exhaustive testing strategy* which may not always be practical. The Derivation Coverage criterion is equivalent to the comprehensive testing of all feature model configurations where all of the strings derivable from a BNF grammar is equivalent to the set of all products of the feature model. To address the practicality aspect of this coverage criterion, the Terminal Symbol and Production coverage criteria were defined.

Definition 2 ***Terminal Symbol Coverage*** *[1] – The* test suite *must contain every terminal symbol of grammar G.*

Definition 3 ***Production Coverage*** *[1] – The* test suite *must contain every production rule of grammar G.*

It is clear that both of these coverage criteria simplify the test suite requirements and are hence more practical than Derivation Coverage. However, from the perspective of testing, these two coverage criteria may be too restrictive; because it is possible to create several very large string productions of the BNF grammar and hence cover all of the terminal symbols using only a very few string products. For instance, for GPL it is possible to create the following two configurations from the feature model EBNF grammar: {`t_Directed, t_Weighted, t_DFS, t_ShortestPath, t_CycleDetection, t_MST, t_StronglyConnected, t_Approximation, t_BruteForce`} and {`t_Undirected, t_unweighted, t_BFS, t_MST`}[3]. These two configurations together are able to cover all of the terminals of the grammar and hence satisfy Terminal Symbol Coverage criterion. Another example is the following configuration: {`t_Undirected, t_Unweighted, t_DFS, t_Approximation`}, which satisfies the Production Coverage criterion since it is created by invoking all of the production rules of the feature model EBNF grammar.

The above examples show that these two coverage criteria are too restrictive for the development of a sufficient number of tests due to the fact that a small number of words (strings) could potentially satisfy their requirements. For this reason, there is a high probability that tests developed based on these criteria would not cover all of the interactions between the features of a feature model. In the following, we develop other coverage criteria to make sure that the developed tests analyze the impact of each individual feature on all other features.

4 Our Approach

The coverage criteria that we propose are devised based on two fundamental postulates:

1. The available features of a feature model can have mutual interactions with each other, which could possibly result in unforseen faults in the final product;

2. The boundary conditions and their corresponding values within equivalence partitions can be considered as hotspot locations for faults.

The implication of the first postulate is that the developed tests should not only analyze each fea-

[3]Order of the terminals is irrelevant in a feature model configuration.

ture individually for fault[4] but they should also consider the interactions between features. For instance, it is possible that feature f_1 does not function only when it is configured alongside feature f_2. Another case would be when feature f_3 fails to perform properly when it is not accompanied in the configuration with feature f_4. Therefore, it is important to develop test suites that are based on productions developed by the enforcement of the inclusion or exclusion of a certain feature. This gives rise to the following two coverage criteria:

Definition 4 ***Feature Inclusion Coverage*** *Let $\mathfrak{F}$ be the set of features in a feature model and $\mathcal{C}_f$ be the set of all configurations of the feature model that can be derived from the EBNF grammar G that include $f \in \mathfrak{F}$. The* test suite *must contain $\forall f \in \mathfrak{F}$ at least one of the members of $\mathcal{C}_f$.*

The main purpose of this coverage criterion is to address individual features and also *feature interactions*. Feature Inclusion Coverage criterion makes sure that all of the features of the feature model are considered one by one and used for configuring the feature model using the feature model EBNF grammar. From the set of possible productions developed by the grammar for each feature, the Feature Inclusion Criterion requires at least one of these productions to be included in the test suite. This criterion is developed to make sure that the effect of the inclusion of each individual feature on the feature model configuration and the other features is covered by the developed tests. The fact that a minimum of one configuration from the set of possible configurations for each feature is selected restricts the coverage of the test suite under this coverage criterion. We will later address this issue using the second postulate based on *boundary value analysis* (Definitions 6 and 7).

Definition 5 ***Feature Exclusion Coverage*** *Let $\mathfrak{F}$ be the set of features in a feature model and $\mathcal{E}_f$ be the set of all configurations of the feature model that can be derived from the EBNF grammar G that do not include $f \in \mathfrak{F}$. The* test suite *must contain $\forall f \in \mathfrak{F}$ at least one of the members of $\mathcal{E}_f$.*

Feature Exclusion Coverage aims at addressing *feature masking* [2]. Essentially, feature masking can happen when the presence of a feature prevents a certain circumstance to happen in another feature. The above criterion attempts to support the development of tests that are able to reveal such situations. For this purpose, for any given feature in the feature model at least one test in the test suite is guaranteed to exist that does not include that feature. This way situations with feature masking are covered. So in summary, the intention of these two coverage definitions are as follows:

- Feature inclusion coverage addresses cases of unintended and undesirable feature interaction and dependencies;
- Feature exclusion coverage supports the identification of feature masking.

Both feature inclusion coverage and exclusion coverage can suffer from the fact that they only require a minimum of one configuration for each feature. We address this issue through the employment of the concepts of *equivalence partitioning* and *boundary value analysis* [28, 27].

The main idea behind equivalence partitioning is to divide the possible test space into segments with similar characteristics from which tests can be derived. Tests are often developed such that at least one test addresses each of the partitions. Furthermore, software testers have come to understand that faults tend to occur more frequently at the boundaries of the test space [1]; therefore, rather than testing random values from anywhere in the test space, it is preferred that tests are designed to cover the boundaries. This is referred to as *boundary value analysis*. Given the fact that equivalence partitioning creates boundaries between the partitions that it develops, boundary value analysis techniques have been traditionally used along with equivalence partitioning to develop suitable tests based on the boundaries of the developed partitions.

We employ the same strategy to overcome the limitation of both of the Feature Inclusion and Feature Exclusion Coverage criteria. In our approach, we consider the set $\mathcal{C}_f$ ($\mathcal{E}_f$) to be the relevant partition pertaining to feature f in the feature model. Now, we would have $|\mathfrak{F}|$ number of partitions, which is equivalent to the number of features available in the feature model. We will assume that all of the configurations in each $\mathcal{C}_f$ ($\mathcal{E}_f$) form an equivalence partition with regards to the requirements of

[4]An *fault* is a bug, error or alike in the implementation of a feature that is provided in the domain engineering phase.

Definitions 4 and 5. Given that the equivalence partitions are created, the boundaries of each partition need to be defined.

In order to define the boundaries of the equivalence partitions, it is important to consider the nature of the values in each partition. In our case, the content of each partition ($\mathcal{C}_f$ or $\mathcal{E}_f$) is a set of feature model configurations developed by the enforcement of the inclusion or exclusion of f. Each $c \in \mathcal{C}_f$ shares the same characteristic of being developed based on the presence or absence of f; therefore, it is reasonable to assume that the boundaries of this partition are the smallest and the largest developed configurations in $\mathcal{C}_f$ ($\mathcal{E}_f$), i.e., $c_u, c_l \in \mathcal{C}_f$ that have the most number of features and the least number of features form the upper and lower boundaries of each partition, respectively. This is meaningful in light of the fact that Batory et al. define each feature as an incremental piece of functionality [6]. It is now possible to formulate appropriate coverage criteria based on the boundaries of the equivalence partitions.

Definition 6 ***Max Feature Inclusion (Exclusion) Coverage*** *Let $\mathfrak{F}$ be the set of features in a feature model and $\mathcal{C}_f$ ($\mathcal{E}_f$) be the set of all configurations of the feature model that can be derived from the EBNF grammar G that do (not) include f. Then $c_u \in \mathcal{C}_f$ ($e_u \in \mathcal{E}_f$) is the upper bound of $\mathcal{C}_f$ ($\mathcal{E}_f$) iff $\nexists c_u^{'} \in \mathcal{C}_f$ ($\nexists e_u^{'} \in \mathcal{E}_f$) s.t. $|c_u| < |c_u^{'}|$ ($|e_u| < |e_u^{'}|$). The* test suite *must contain all upper bounds in $\mathcal{C}_f$; $\forall f \in \mathfrak{F}$ ($\mathcal{E}_f$; $\forall f \in \mathfrak{F}$).*

Definition 6 provides the basis for the development of two coverage criteria which require the inclusion of the upper bound of the partitions developed based on the inclusion or exclusion of the available features. Two other coverage criteria can be developed based on the lower bounds of the equivalence partitions.

Definition 7 ***Min Feature Inclusion (Exclusion) Coverage*** *Let $\mathfrak{F}$ be the set of features in a feature model and $\mathcal{C}_f$ ($\mathcal{E}_f$) be the set of all configurations of the feature model that can be derived from the EBNF grammar G that do (not) include f. Then $c_l \in \mathcal{C}_f$ ($e_l \in \mathcal{E}_f$) is the lower bound of $\mathcal{C}_f$ ($\mathcal{E}_f$) iff $\nexists c_l^{'} \in \mathcal{C}_f$ ($\nexists e_l^{'} \in \mathcal{E}_f$) s.t. $|c_l| > |c_l^{'}|$ ($|e_l| > |e_l^{'}|$). The* test suite *must contain all lower bounds in $\mathcal{C}_f$; $\forall f \in \mathfrak{F}$ ($\mathcal{E}_f$; $\forall f \in \mathfrak{F}$).*

Together these four coverage criteria provide the means for covering all of the boundary value conditions of the equivalence partitions. So, we can assume that any test generation strategy that satisfies these four coverage criteria would be able to identify some of the most significant faults caused by one of the following: 1) fault in each individual feature; 2) fault caused by the interactions between features; and 3) fault caused by the masking of features.

Now, since our feature models are expressed through EBNF grammars, we need to distinguish between terminals and non-terminals in our coverage criteria. In other words, there needs to be a distinction between the leaf features and the non-leaf features of the feature model. This is because non-terminals have much more impact on the possible derivations of the EBNF grammar. For instance, the inclusion or exclusion of a non-terminal will more greatly limit the possible productions of the grammar rather than the inclusion or exclusion of a terminal. The discernment between terminals and non-terminals will further refine the four coverage criteria defined in Definitions 6 and 7 into eight coverage criteria, namely 1) Maximum Terminal Inclusion coverage (MxTI), 2) Maximum Non-Terminal Inclusion coverage (MxNTI), 3) Maximum Terminal Exclusion coverage (MxTE), 4) Maximum Non-Terminal Exclusion coverage (MxNTE), 5) Minimum Terminal Inclusion coverage (MinTI), 6) Minimum Non-Terminal Inclusion coverage (MinNTI), 7) Minimum Terminal Exclusion coverage (MinTE) and 8) Minimum Non-Terminal Exclusion coverage (MinNTE).

The interpretation of each of these coverage criteria can be done based on Definitions 6 and 7. For instance, MxTI is a form of Max Feature Inclusion Coverage that requires the test suite to include all the upper bounds of the partitions that have been produced by the enforcement of the inclusion of each terminal of the EBNF grammar (leaf feature of the feature model) or similarly, MinNTE is a case of Min Feature Exclusion Coverage, which specifies that the test suite must include all of the lower bounds of the partitions that have been created by the exclusion of each individual non-terminal in the EBNF grammar (non-leaf features).

It is important to note how equivalence partitioning and boundary value analysis are relevant for our purpose given that they have only been used in the

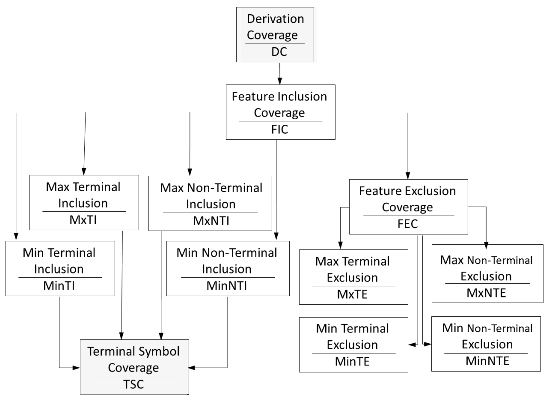
Derivation Coverage DC
Feature Inclusion Coverage FIC
Max Terminal Inclusion MxTI
Max Non-Terminal Inclusion MxNTI
Feature Exclusion Coverage FEC
Min Terminal Inclusion MinTI
Min Non-Terminal Inclusion MinNTI
Max Terminal Exclusion MxTE
Max Non-Terminal Exclusion MxNTE
Min Terminal Exclusion MinTE
Min Non-Terminal Exclusion MinNTE
Terminal Symbol Coverage TSC

Figure 2: Criteria subsumption relations.

Algorithm 1: Maximum Terminal Inclusion (MxTI)

```
input  : A Feature Model $\mathfrak{F}$
output: A Test Suite $\Phi$
$\Phi \leftarrow \emptyset$;
foreach $f \in \mathfrak{F}$ do
    if IsLeaf($f$) then
        temp ←generateLgtConf($f$,$\mathfrak{F}$);
        if temp $\notin \Phi$ then
            $\Phi \leftarrow \Phi \cup$ temp;
```

Table 1: The Objects of Study.

Feature Model	NF	CTCR	NVC
Digital Video System	26	23%	22,680
Bicycle	27	14%	1,152
ATM Software	29	0	43,008
TV-Series Shirt	29	27%	21,984
Smart Home	35	0	1,048,576
Sienna	38	26%	2,520
Arcade Game	61	55%	3.3E+09
HIS	67	11%	6,400
Model Transformation	88	0	1.65E+13

literature over the inputs of a single software and not for test generation for a family of systems. We note that the main idea is to find *corner cases*. The reason that minimum and maximum coverage criteria define such corner cases relates to *variability* in software product lines. In other words, the tests selected in the minimum criteria equate to the set of mandatory features with the inclusion of 'NO' optional features; whereas, the tests selected in the maximum criteria involves the inclusion of all possible variable parts of the product line (optional features). Therefore, maximum and minimum coverage criteria provide a basis for testing boundaries of the configuration space based on variation points in the product line and hence represent valid partitions for performing boundary value analysis considering variability in the product line.

Figure 2 shows the subsumption relationships between the coverage criteria and their relation to derivation coverage proposed by Ammann and Offutt [1]. It is clear that Derivation Coverage subsumes all other criteria since it basically consists of all the possible productions of the grammar. Furthermore, Feature Inclusion Coverage subsumes Feature Exclusion Coverage since there may be cases in the latter where a given feature is never seen; but Feature Inclusion Coverage is guaranteed to cover all of such cases. Finally, the set of terminal inclusion-based coverage criteria are guaranteed to subsume Terminal Symbol Coverage based on Definition 4, which will ensure that all leaf features are included in the test suites; however, this is not the case for the other coverage criteria.

It is now quite straightforward to generate test suites that conform to these coverage criteria by generating the possible minimum/maximum length productions of the grammar that include or exclude a given non/terminal. Algorithm 1 shows the details of test suite generation based on MxTI. The other seven test suite generation strategies based on the coverage criteria are quite similar.

As seen in Algorithm 1, the MxTI strategy for test suite generation will try to develop test suites that include the largest feature model configurations that include each one of the leaf features in the feature model. For this purpose, the algorithm first generates the largest configuration that contains a given feature (`generateLgtConf`). The generation of the largest (smallest) feature model configuration can be easily done using a weighted context-free grammar (weighted EBNF) [14]; this is equivalent to the generation of the longest (shortest) string that contains a given terminal. If this largest configuration, which is the potential test, does not already exist in the test suite, it will be added; otherwise, it will be discarded. This process is repeated for all leaf features. Once all features are processed, Φ is the final test suite that includes the tests generated based on the Maximum Terminal Inclusion (MxTI) strategy that need to be tested. The other seven strategies are implemented similarly.

5 Evaluation

5.1 Objects of Analysis

For the purpose of our experiments, we have selected a set of feature models that are publicly available through the SPLOT website [23]. SPLOT's goal is to facilitate the transition process from research to practice and therefore provides the means for both researchers and practitioners to share and contribute their software product line feature models in an open and free manner. These feature models are expressed in the SXFM format.

The feature models that were used in our experiments with three of their important metrics are shown in Table 1. The NF, CTCR, and NVC metrics denote the number of features in the feature model, the ratio of the number of distinct features in the integrity constraints to the number of feature model features, and the number of valid configurations of the feature model, respectively. Feature models with a CTCR of zero do not consist of any integrity constraints.

5.2 Experiment Design

In order to evaluate the effectiveness of the proposed approach, we have performed our experiments on the feature models introduced in Table 1 using an automated software program. The program is able to parse feature models defined in SXFM and automatically develop the corresponding EBNF grammar for that feature model. Given the conversion of the feature models into EBNF grammar, a set of test suite generation programs were developed for each one of the eight coverage criteria, each of which generates suitable test suites consisting of feature model configurations, i.e. the tests, based on the EBNF grammar production rules. Each test in the generated test suites is a derived software application (product) that needs to be tested. So assuming that the size of a test suite is $\mathcal{S}$ and it takes $O(m)$ to test each individual application, the total complexity of testing the feature model is now $O(\mathcal{S} \times m)$ as opposed to $O(2^n \times m)$.

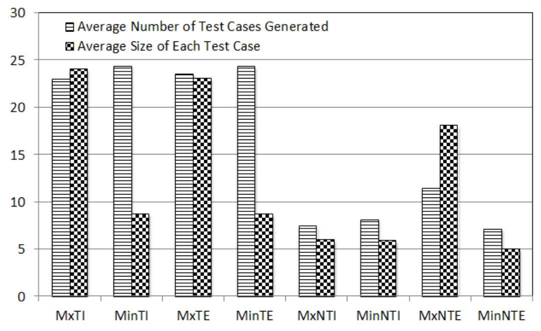

Figure 3: The size of the test suites.

Therefore, as long as $\mathcal{S} \lll 2^n$, we are successful in reducing the effort needed for testing the product line feature model to $O(m)$ – due to a negligible $\mathcal{S}$. Based on this, we explore the following two hypotheses:

H1 Our test suite generation strategies that are based on the eight coverage criteria are able to develop test suites such that $\mathcal{S} \lll 2^n$;

H2 Although $\mathcal{S} \lll 2^n$, the generated test suites are able to maintain an acceptable (high) fault coverage.

Now in practice, once a feature model is configured and a specific application is derived, the selected features will be replaced by suitable software components, Web Services or software programs that are able to fulfil the requirements of that feature. The replacement of features with appropriate implementation details is often done manually by engineers using proprietary software; therefore, information on them is not publicly available. In view of this issue and to be able to evaluate the efficacy of our coverage criteria and their corresponding test suite generators, we developed an *fault generation simulator*. The lack of publicly available datasets has already been pointed out in [10, 4, 3] and several authors and tool suites (such as FaMa [7] and 3-CNF model generator [23]) have been using model generation techniques to test their work for the lack of a better means.

The simulator considers the three introduced metrics shown in Table 1 for each feature model, i.e., NF, CTCR, and NVC, and generates faults for the feature model correspondingly. Feature models with more features, higher ratio of CTCR and a

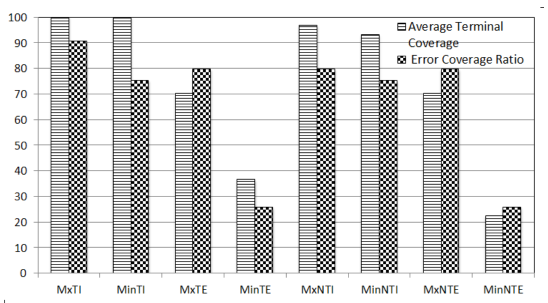

Figure 4: Coverage of generation strategies.

higher number of valid configurations will contain more faults generated by the simulator. The generated faults are in one of the following categories:

1. *faults in individual features*: The simulator will select a number of features from the feature model proportional to NF and NVC. These selected individual features will be considered to contain faults and will need to be detected by the generated test suites;

2. *faults in repulsive features*: These faults will be generated in the form of n-tuples, where each tuple contains a set of 2 or more features. The idea behind these faults is that a configuration will contain fault due to the interactions between the features if the n features in the n-tuple are all in that configuration. These faults are proportional to NF and CTCR, i.e., more faults in repulsive features would be generated for a feature model with higher values for NF and CTCR.

3. *faults in attracting features*: For this type of fault, a pair of features are selected and an fault occurs if one of the features in the pair is present in the configuration and the other is not. Similarly, the number of faults in attracting features is dependent on NF and CTCR.

Further details and code for the fault generator is available for download at `http://ebagheri.athabascau.ca/splt/splt.zip`. Interested researchers are encouraged for replication studies.

5.3 Results and Analysis

In order to evaluate the test suite generation strategies, the fault generation simulator was executed over each of the feature models in Table 1. This resulted in a set of faults for each of the feature models, which needed to be identified and covered by the generated test suites. The composition of the generated test suites were as follows: each test suite consisted of a set of tests, which were feature model configurations (complete productions of the context-free grammar) that corresponded with the eight coverage criteria. Each of the feature model configurations are therefore a collection of selected features from the feature model. The obtained raw data (non-average values) are reported at `http://ebagheri.athabascau.ca/splt/appendix.pdf` due to space limitation. The following analyzes the observations in detail.

Figure 3 shows the size of the generated test suites. This figure depicts two aspects of the test suite size, i.e., the average number of tests generated for each test suite and also the average size of each test, i.e., the number of features that it contains. The average values are calculated over the values obtained from each of the test suites generated for each of the objects of study over 10 trials to remove the effect of randomness in the fault generator. Two main observations can be made from Figure 3: 1) test suites generated based on EBNF grammar terminals tend to be larger both in the number of tests that they offer and also the number of features that each test contains. This can be taken as a sign of comprehensiveness; however, before this conclusion can be made, it is important to evaluate whether the greater number of tests in each test suite results in a higher fault coverage or not; 2) the average size of the tests (the number of features that it contains) in the suites that were generated by the lower bound of the equivalence partitions are smaller than those generated based on the upper bound, which is a logical consequence of these coverage criteria.

It was observed that test suites generated based on terminals tend to be larger and may hence be more comprehensive. It is important to analyze whether the larger size of the test suites developed based on this strategy constitutes higher coverage or not. For this purpose, Figure 4 shows two important results of the test suites, namely the *average*

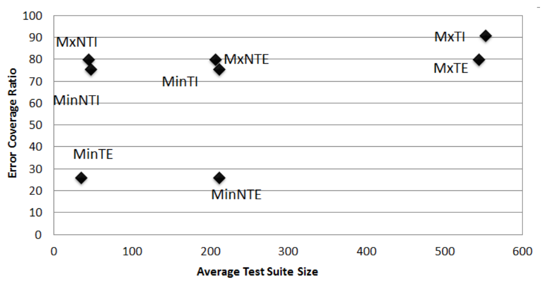

Figure 5: efficacy of generation strategies.

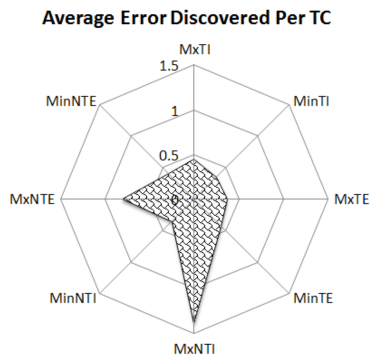

Figure 6: Average fault detected per test.

terminal coverage of the test suites and the *fault coverage ratio*. Average terminal coverage shows the percentage of the number of distinct features in the tests of the test suite that could appear in a feature model configuration over all of the available leaf features in the feature model. The fault coverage ratio is the number of faults covered by the test suite over all of the existing faults. It was argued earlier that MxTI and MinTI subsume Terminal Symbol Coverage (c.f. Figure 2). This is reflected in Figure 4 as well where the average terminal coverage is 100%. It is worth mentioning that MxNTI and MinNTI have a very high average terminal coverage, an indication that these two coverage criteria are able to cover most of the leaf features of the feature model. Moreover, as it can be seen in Figure 4, MxTI and MxNTI have the highest fault coverage ratio. This is an indication that these two coverage criteria are able to detect the most number of faults and are the two test suite generation strategies with the highest fault coverage.

The other important aspect of the test suite generation strategies is their *efficacy*. We view efficacy as the tradeoff between the size and number of the test suites and the fault coverage ratio. Simply put, test suites with a smaller size and fewer number of tests that have a higher fault coverage ratio are more desirable than the others and are hence more *efficient*. We argue that a strategy that is able to generate such test suites is an efficient strategy.

To analyze the efficacy of the test suite generation strategies, the diagram in Figure 5 is plotted, which shows the tradeoff between the average size of the test suites (average number of tests $\times$ average size of tests) and the fault coverage ratio. The most desirable strategies are those that are located in the top left corner of this diagram. This is because the top left corner locates the strategies that have the highest fault coverage ratio and at the same time the smallest test suite size. Based on this figure, MxNTI is the most efficient strategy. Despite the fact that MxTI had a high fault coverage ratio, it is not as efficient as MxNTI because its average test suite size is comparatively higher than MxNTI, meaning that more tests are generated in the test suites of MxTI and therefore much more effort is required as compared to MxNTI to identify the available faults. This has also been portrayed in Figure 6, which depicts the average number of faults detected per test developed by each of the strategies. As it can be seen, MxNTI has the highest detection rate per test. This is an observation that depicts how MxNTI is more efficient than the other strategies.

In summary, the results of our analyis show that MxNTI and MxTI have the highest fault coverage and MxNTI has the highest efficacy from amongst the eight coverage criteria. In any case, since both of these strategies have significantly reduced the number of tests as compared to the standard strategy based on Derivation Coverage, both of these strategies can be considered as being viable for generating test suites for feature models. However, in more resource constraint environments, MxNTI has benefits over MxTI with the trade-off of a slightly lower fault coverage.

Now that we have analyzed each of the test suite generation strategies in isolation, it is important to see how the collection of the test suites perform together, i.e., what would the results of the tests be if all of the test suites generated by the eight coverage criteria were combined into one test suite (removing the duplicate tests). The results of this

Table 2: Total number of tests generated by the coverage criteria and the related fault coverage.

	NF	#Tests by derivation coverage	#Tests by our approach	fault coverage of our approach
Digital Video System	26	2.27E+04	63	89%
Bicycle	27	1152	63	87%
ATM Software	29	4.30E+04	81	78.33%
TV Series	29	21984	103	84.00%
Smart Home	35	1.05E+06	125	92.30%
Sienna	38	2520	126	97%
Arcade	61	3.30E+09	201	92.33%
HIS	67	6400	147	96.23%
Model Transformation	88	1.65E+13	255	92%

is reported in Table 2. The results shown in Table 2 can be used to evaluate our two hypotheses (H1 and H2). It is important to compare our work with tests generated by derivation coverage as it has 100% fault coverage. As can be seen, the number of tests generated by all of the eight test suite generators is significantly less than the total number of possible tests in a comprehensive testing process based on derivation coverage. Therefore, we can claim that H1 is correct in that $\mathcal{S} \lll 2^n$ (e.g., 63 tests vs $2.27E + 04$ tests – first row in Table 2). Furthermore, the second hypothesis (H2) that claimed the generated test suites are efficient by having a high fault coverage can also be considered to be true given the fact that in the worst case, the fault coverage is 78.33%, which is relatively high given the significant reduction in the size of tests. The average fault coverage over all the feature models is close to 91%. Based on the observations of Table 2, we believe that compared to the comprehensive tests based on derivation coverage criterion, our test suite generation strategies, which are based on the eight context-free grammar-based coverage criteria are able to generate small size test suites while maintaining a high fault coverage. Given that we did not have access to the programs of other test generation techniques, our comparison has been with baseline (derivation coverage). In the future we will look into further comparative analytics as the code for other related work become publicly available.

5.4 Threats to Validity

We identify three main sources of threat to the validity for our experiments that need to be pointed out and clearly addressed. The first issue relates to *external validity*, which is the extent to which the obtained results of a study can be generalized to settings other than that under study and other relevant research scenarios. In our experiments, a limited number of feature models from the SPLOT repository were used due to our restricted access to appropriate models within the area of software product lines. Even among the feature models in SPLOT, many of them were too small or non-descriptive to be useful, and therefore, only a limited number could be used in our experiments. Although these numbers are comparable to (even higher than) similar studies in the area of product lines [20, 10, 25], their limited number (not representative of all possibilities) may pose threats to the generalizability of the drawn conclusions. We are currently working on the collection of a larger set of feature models for future studies. The second issue is again related to external validity. In the experiments we have relied on a fault taxonomy introduced earlier in the paper. However, we do not claim this to cover all types of faults that can happen in software product line feature models. Therefore, given the limited scope of our fault types, the results cannot be generalized for other types of faults that could be encountered and are not covered. At the present time, a comprehensive fault taxonomy is yet to be developed for software product lines; therefore, our focus has been on three dimensions that were introduced in the paper and the results are valid for these fault types only. The last issue again concerns the involvement of the test generator simulator that was used in our experiments. Given that actual implementation components for each feature in the product line are not released by the industry, the actual statistical distribution of fault in features is not accurately known and hence in our approach, heuristics-based fault generation has been used. As such statistics are not accessible to the research community yet, we believe that the development of this simulator is a significant step for reproducible test generation strategy evaluation, which can be gradually improved by the community as more insight is gained through further replication studies.

6 Related Work

There are only a limited number of approaches that directly address the issue of test generation in software product lines. Closest to our work is by Cohen et al [10]. The authors propose to map OVM product line representation models onto a relational model for defining the cumulative coverage criteria based on whose combination with combinatorial interaction testing methods suitable tests are generated. The main drawback of this work is that it has not been empirically validated. Similarly, Cabral et al. suggest the development of a underlying representation called feature dependency graphs [8]. This representation is later used in combination with a graph based testing approach called the FIG basis path method. The work reports a high fault coverage for tests as small as 24% of the application space. However, the largest model used in the experiments consisted of 38 products, which is not significant compared to our experiments. Given the complexity and number of possible configurations, some other techniques employ automatic analysis based on SAT solvers [19] such as Alloy [16]. For instance, Uzuncaova et al. [31] propose a hybrid approach by combining methods from software product lines and specification-based testing using Alloy. In their approach, each product is defined as a composition of features represented as Alloy formula. The Alloy analyzer is used to incrementally generate test cases over partial specifications of the products. This approach is an improvement over previous work that generated test cases in a single pass execution of Alloy over complete specifications [30]. Also, Perrouin et al. employ the concept of T-wise test generation [25]. Their approach attempts to address the large combinatorial number of required tests for a software product line by only generating test sets that cover all possible T feature interactions. To achieve this, the authors devise strategies to disintegrate T-wise combinations into smaller manipulable subsets, which are then used in a developed toolset over Alloy for generating the tests.

The authors of [20] base their work on the assumption that despite the number of product line configurations being exponential, but features are often *behavior-irrelevant*, i.e., they augment but do not change the behavior of a system. According to this assumption, many of the test cases become overlapping and the smaller test sets will be redundant; therefore, the authors are able to design a static program analysis method to find the behavior-irrelevant features of the product line and hence reduce the size of the test space. Unlike the works presented in [20, 25, 30] that generally focus on syntactical aspects of product lines for generating test cases and employ strategies such as pairwise testing, our approach employs a semantic approach for test generation where candidate tests are developed based on a clear set of coverage criteria. Hence, using our proposed approach the software tester is able to generate tests based on specific criteria that match her intent and hence is able to interpret the test outcomes in light of the selected test coverage criteria.

Other work which reduce the test space also exist that mainly focus on the use of user requirements to identify the most important set of features that need to be tested [29]. Two recent systematic studies by Neto et al. [13] and Engstrm and Runeson [15] cover the current state-of-the-art in software product line testing.

7 Conclusions

In this paper, we have proposed to represent product line feature models through context-free grammars and defined eight coverage criteria for product lines building on the work by Ammann and Offut on BNF coverage criteria [1]. These coverage criteria are based on the shortcomings of the existing coverage criteria for BNF grammars for the purpose of testing product line feature models. Our work is one of the first in the field that proposes clearly defined and validated coverage criteria for testing software product lines. Equivalence partitioning and boundary value analysis have been the basis for defining the coverage criteria. Furthermore, the proposed criteria have been used as a foundation for defining several test suite generation strategies. In order to evaluate the test suite generation strategies, several experiments were conducted over SPLOT software product line feature models developed by both researchers and practitioners. The proposed strategies have shown to be quite efficient for substantially reducing the number of required tests and also having high fault detection ratio. We evaluated two main hypotheses showing that the test suite generation strategies based on the eight coverage criteria are able to reduce the size of the test space while maintaining a high fault coverage.

References

[1] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge Univ Pr, 2008.

[2] P. Ammann, J. Offutt, and H. Huang. Coverage criteria for logical expressions. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, page 99. Citeseer, 2003.

[3] E. Bagheri, F. Ensan, and D. Gasevic. Decision support for the software product line domain engineering lifecycle. *Automated Software Engineering (DOI: 10.1007/s10515-011-0099-7)*, pages online–first, 1–43, 2012.

[4] Ebrahim Bagheri and Dragan Gasevic. Assessing the maintainability of software product line feature models using structural metrics. *Software Quality Journal*, 19:579–612, September 2011.

[5] Ebrahim Bagheri, Tommaso Di Noia, Dragan Gasevic, and Azzurra Ragone. Formalizing interactive staged feature model configuration. *Journal of Software Maintenance and Evolution: Research and Practice*, page to appear, 2011.

[6] Don Batory, David Benavides, and Antonio Ruiz-Cortes. Automated analysis of feature models: challenges ahead. *Commun. ACM*, 49:45–47, December 2006.

[7] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortes. FAMA: Tooling a framework for the automated analysis of feature models. In *Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*, pages 129–134, 2007.

[8] Isis Cabral, Myra Cohen, and Gregg Rothermel. Improving the Testing and Testability of Software Product Lines. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines: Going Beyond*, volume 6287 of *Lecture Notes in Computer Science*, pages 241–255–255. Springer Berlin / Heidelberg, 2011.

[9] P. Clements and L. M. Northrop. Software product lines, visited june 2009, http://www.sei.cmu.edu/programs/pls/sw-product-lines_05_03.pdf, 2003.

[10] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Coverage and adequacy in software product line testing. In *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, ROSATEA '06, pages 53–63, 2006.

[11] K. Czarnecki and U. Eisenecker. *Generative programming*. Springer, 2000.

[12] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration Using Feature Models. In *Proceedings of the Third Software Product Line Conference 2004*, pages 266–282. Springer, LNCS 3154, 2004.

[13] Paulo Anselmo da Mota Silveira Neto, Ivan do Carmo Machado, John D. McGregor, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. A systematic mapping study of software product lines testing. *Inf. Softw. Technol.*, 53:407–423, May 2011.

[14] Manfred Droste, Werner Kuich, and Heiko Vogler. *Handbook of Weighted Automata*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[15] Emelie Engström and Per Runeson. Software product line testing - a systematic mapping study. *Inf. Softw. Technol.*, 53:2–13, January 2011.

[16] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):290, 2002.

[17] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, A.S. Peterson, and CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST. *Feature-oriented domain analysis (FODA) feasibility study*. Carnegie Mellon University, Software Engineering Institute, 1990.

[18] KC Kang, J. Lee, and P. Donohoe. Feature-oriented product line engineering. *IEEE software*, 19(4):58–65, 2002.

[19] S. Khurshid and D. Marinov. TestEra: Specification-based testing of Java programs using SAT. *Automated Software Engineering*, 11(4):403–434, 2004.

[20] C.H.P. Kim, D.S. Batory, and S. Khurshid. Reducing combinatorics in testing product lines. In *Proceedings of the tenth international conference on Aspect-oriented software development*, pages 57–68. ACM, 2011.

[21] Kwanwoo Lee, Kyo Chul Kang, and Jaejoon Lee. Concepts and guidelines of feature modeling for product line software engineering. In *International Conference on Software Reuse*, ICSR-7, pages 62–77, 2002.

[22] Roberto E. Lopez-Herrejon and Don S. Batory. A standard problem for evaluating product-line methodologies. In *Proceedings of the Third International Conference on Generative and Component-Based Software Engineering*, pages 10–24, 2001.

[23] Marcilio Mendonca, Moises Branco, and Donald Cowan. S.p.l.o.t.: software product lines online tools. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 761–762, New York, NY, USA, 2009. ACM.

[24] G.J. Myers, C. Sandler, and T. Badgett. *The art of software testing*. Wiley, 2011.

[25] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In *ICST 2010*, pages 459–468, 2010.

[26] K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

[27] Muthu Ramachandran. Testing software components using boundary value analysis. *EUROMICRO Conference*, 2003.

[28] Stuart C. Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. *Software Metrics, IEEE International Symposium on*, 1997.

[29] Kathrin D. Scheidemann. Optimizing the selection of representative configurations in verification of evolving product lines of distributed embedded systems. In *Proceedings of the 10th International on Software Product Line Conference*, pages 75–84, 2006.

[30] Engin Uzuncaova, Daniel Garcia, Sarfraz Khurshid, and Don S. Batory. A specification-based approach to testing software product lines. In *ESEC/SIGSOFT FSE*, pages 525–528, 2007.

[31] Engin Uzuncaova, Sarfraz Khurshid, and Don S. Batory. Incremental test generation for software product lines. *IEEE Trans. Software Eng.*, 36(3):309–322, 2010.