Evolutionary Search-based Test Generation for Software Product Line Feature Models

Faezeh Ensan, Ebrahim Bagheri, and Dragan Gasevic

University of British Columbia, Athabasca University

Abstract. Product line-based software engineering is a paradigm that models the commonalities and variabilities of different applications of a given domain of interest within a unique framework and enhances rapid and low cost development of new applications based on reuse engineering principles. Despite the numerous advantages of software product lines, it is quite challenging to comprehensively test them. This is due to the fact that a product line can potentially represent many different applications; therefore, testing a single product line requires the test of its various applications. Theoretically, a product line with n software features can be a source for the development of 2^n application. This requires the test of 2^n applications if a brute-force comprehensive testing strategy is adopted. In this paper, we propose an evolutionary testing approach based on Genetic Algorithms to explore the configuration space of a software product line feature model in order to automatically generate test suites. We will show through the use of several publicly-available product line feature models that the proposed approach is able to generate test suites of $O(n)$ size complexity as opposed to $O(2^n)$ while at the same time form a suitable tradeoff balance between error coverage and feature coverage in its generated test suites.

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 many applications of a target domain [5, 22]. 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 [14, 11]. Therefore, a software product line for a given domain is a model of that domain such that it formalizes the existence of and the interaction between all of the possible features of the domain.

Software product lines are often represented through *feature models*, which are tree-like structures whose nodes are the domain features and the edges are the interaction between the features. Each feature model is an abstract representation of the possible applications of a target domain; therefore, it is possible to derive 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*. 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 domain-dependent applications.

As a matter of fact, the number of possible configurations of a feature model increases *exponentially* with the size of the feature model, i.e., a feature model with more features is a potential for many more configurations [13]. 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 the analysis of one single application that needs to be fully evaluated. However, the process of testing a software product line and ensuring that most if not all of its errors[1] are detected requires comprehensive analysis of all of its potential products, i.e., as opposed to testing a single application, testing a product line requires testing multiple applications and is therefore much more time consuming and costly. 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 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 development of test generation strategies that would create small but efficient tests for testing large product lines.

In many cases, an acceptable tradeoff between error coverage and the number of generated tests would need to be found such that while a high error coverage is reached, the number of tests that are generated for the product line are kept as small as possible. In this paper, we benefit from Genetic Algorithms 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 of testing all of the possible applications of the product line. Throughout this paper, a *test suite* is a restricted collection of applications derived from a feature model in order to reduce the test space; hence a test suite consists of several applications from all possible 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 instead of testing all of the application space. Therefore, simply put, our goal is to select a set of applications referred to as the test suite where each individual application in the test suite is called a test. It should be noted that in reality the testing of each test would require a set of test cases. The generation of the test cases for each application can be done using conventional test case generation techniques for single software applications and is outside the scope of this paper. Here we will only focus on the generation of test suites for software product lines.

To this end, we look at *search-based test generation* to explore the feasible test space and find appropriate test suites for a given software product line. More specifically, we employ Genetic Algorithms for this purpose. Genetic Algorithms allow us to start with a randomly generated set of tests for a product line and gradually improve the quality of the tests by evolving them in a controlled process. In this paper, we will discuss how a test generation process can be defined based on Genetic Algorithms for software product lines. This will include the process of designing appropriate *chromosome* representations of software product line configurations and the definition of a suitable *fitness function*. It will be shown through our experimentations that the proposed strategy is able to generate small but efficient test suites for software product lines.

The rest of the paper is organized as follows: the next section covers the background regarding the structure of software product line feature models and also a brief introduction to search-based test generation and genetic algorithms. The details of our proposed Genetic Algorithm based test generation process is formalized and introduced in Section 3. Section

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

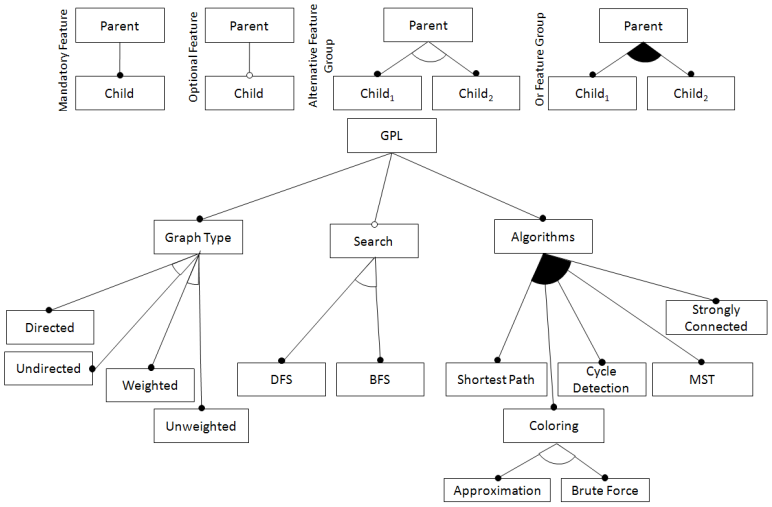

Fig. 1. Graph product line feature model.

4 details the evaluation of the proposed test generation process. This is followed by the review of the relevant literature in Section 5. The paper is then concluded in Section 6.

2 Background

2.1 Feature Models

Feature models are one of the most important modeling means for representing aspects of a software product line. In this paper, we focus on feature models to deal with the details of a product line. Within the context of feature models, features are important distinguishing aspects, qualities, or characteristics of a family of systems [14]. 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 root node represents a domain concept and the other nodes and leafs illustrate the features.

In a feature model, features are hierarchically organized and are typically classified as: *Mandatory*, the feature must be included in the description of its parent feature; *Optional*, the feature may or may not be included in its parent description given the situation; *Alternative feature group*, one and only one of the features from the feature group can be included in the parent description; *Or feature group*, one or more features from the feature group can be included in the description of the parent feature.

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: *Includes*, the presence of a given feature (set of features) requires the *existence* of another

feature (set of features); *Excludes*, the presence of a given feature (set of features) requires the *elimination* of another feature (set of features).

The Graph Product Line (GPL) depicted in Figure 1 is the typical 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 in the form of integrity constraints. Two examples of these constraints are: Cycle Detection EXCLUDES BFS and Strongly Connected INCLUDES DFS.

These 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 its possible configurations. In other words, each possible configuration is a *test* for evaluating the feature model. Taking this approach for testing the simple graph product line feature model shown in Figure 1 would require 308 different applications to be tested (308 tests that need to be evaluated).

2.2 Metaheuristic Search

The generation of tests is often a part of the responsibilities of the test engineer in industrial settings. However, this process is too costly, labor-intensive and lengthy for large applications. For this reason, *metaheuristic search techniques* have been favored by many for the automatic generation of appropriate tests. According to McMinn [15], metaheuristic search techniques are high-level frameworks which utilize heuristics to find rather optimal solutions to combinatorial problems at an acceptable computational cost. Several metaheuristic techniques have been used in software test generation, including *Hill climbing*, which is a local search algorithm; *Simulated annealing*, which is a probabilistic and more flexible extension of Hill climbing; and *Evolutionary algorithms*.

The application of evolutionary algorithms such as Genetic Algorithms to software test generation is referred to as *evolutionary testing* [15]. Evolutionary testing approaches have proven to be effective in generating test input data for specific execution paths or certain program structures [27, 4, 30]. In this paper, we will be exploiting Genetic Algorithms for automatically generating efficient tests for software product line feature models.

Evolutionary algorithms [6] are inspired by *biological evolution*, which includes reproduction, mutation, recombination, and selection; hence, they focus on evolution as a search strategy. The main idea behind this class of metaheuristic search techniques is that given an initial set of possibly random solutions for an optimization problem, an evolutionary algorithm will be able to find a near optimal solution by gradually mutating and recombining these existing solutions into newer solutions such that a measure of fitness is improved in the process. Genetic Algorithms [10] are one of the widely used forms of evolutionary algorithms. They operate by forming a correspondence between the genetic structure of chromosomes and the encoding of solutions in the optimization problem. With this correspondence, since solutions of the optimization problem are represented as chromosomes, natural biological evolution operators such as *mutation* and *cross-over* can be used to generate new chromosomes (i.e., new solutions).

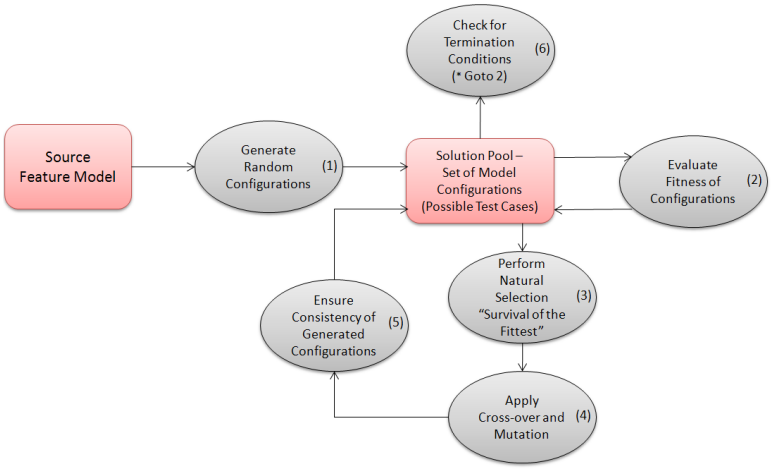

Fig. 2. The overview of our proposed approach.

Essentially, the process of generating new chromosomes based on the previous chromosomes is repeated until an acceptable set of answers are developed. The suitability of each chromosome (solution) for the given optimization problem is evaluated by a *fitness function*. In the Genetic Algorithm process, each round of generating new chromosomes is referred to as a *generation*. The important factor that drives Genetic Algorithms towards a near optimal solution is the role of *natural selection* in each generation, i.e., in each generation weaker chromosomes die (weaker solutions are deleted), and stronger chromosomes survive (solutions with higher fitness are kept) and are moved onto the next generation. With this *survival of the fittest* strategy, Genetic Algorithms often converge to a (near) optimal solution for optimization problems.

3 Test Suite Generation

3.1 Approach Overview

The main objective of our work is to generate small but efficient test suites for software product line feature models. We employ Genetic Algorithms to search the possible configuration space of a feature model (2^n possible feature model configurations) in order to select a limited number of the most effective and covering configurations to serve as the tests in our test suite. The outline of our approach is depicted in Figure 2 and is as follows:

In the first step (1), a set of feature model configurations are randomly generated from the feature model without considering any extra constraints on the feature model other than the fact that the configurations must be valid. These configurations serve as the initial population for Genetic Algorithm. Once the initial population is formed, each of the configurations (aka solutions) is evaluated in step (2) to see how *fit* it is. We will formally define the fitness function in the next section. Given the fitness of the configurations, step (3) discards the weaker configurations, while the rest of the configurations are preserved to be used as seeds for the mutation and cross-over operators in step (4). It is important to mention that since the new population is generated based on the previous configurations, the new configurations in the population are not necessarily always valid. Therefore, step

(5) evaluates the validity of the generated configurations from step (4) and only accepts valid ones[2]. Step (2)-(5) are repeated until the termination condition (introduced later) is satisfied. At the end of this process, the final population of the Genetic Algorithm is considered to be the generated *test suite* and each of the configurations in the population is a *test* in the test suite that can be used as a sample application to be tested for testing the software product line.

3.2 Approach Realization

The representation of possible solutions in a Genetic Algorithm are called chromosomes. Each chromosome is composed of smaller building blocks called genes. In each optimization round of a Genetic Algorithm, a collection of chromosomes are formulated and evolved in order to guide the optimization process. As we will show later, once the algorithm ends, one or several of the chromosomes will represent the possible solutions. In our settings, for a feature model of size n (feature model with n features), we create binary chromosomes of size n. This means that the chromosomes of our Genetic Algorithm are in the form of an array of length n, whose elements can either be 0 or 1. Each gene of the chromosomes shows whether the corresponding feature is selected or not. For instance, assume that a feature model with four features is being tested. A possible chromosome such as $< 0, 1, 1, 0 >$ would represent a configuration of that feature model where the second and third features of the feature model are present in the configuration, while the first and last features are not included. Also, the size of the chromosomes is n because the largest configuration that can be developed could at most contain all of the features, and since the chromosomes represent the possible configurations, we generate chromosomes with n genes.

Each of the chromosomes represents a specific feature model configuration and is therefore considered to be a potential test. Further, the Genetic Algorithm population is viewed as the test suite. With this assumption, the size of the test suites is dependent on the population size selected for the Genetic Algorithm. Some techniques have already been proposed including our own strategy [1] for dynamically configuring the optimal population size for a Genetic Algorithm; however, in this paper, we have chosen to evaluate different population sizes because we are interested in evaluating *feature coverage* as well as *error coverage*. A dynamic approach to determining the population size would have prevented us from studying the tradeoff between feature and error coverage.

In a previous study [2], we have already explored some of the structural properties of feature models and their important characteristics. Two of the most important distinguishing aspects of feature models is their ability to 1) capture variability in design and 2) represent feature interdependencies through integrity constraints. For this reason, we base the Genetic Algorithm fitness function on these two important aspects: variability and integrity constraints. To develop a concrete measurable stance for these aspects we benefit from two metrics for each feature model configuration: 1) *variability coverage* ($\mathcal{VC}$) and 2) *cyclomatic complexity* ($\mathcal{CC}$). In order to compute variability coverage of a product, we count the number of variation points that had to be bounded in order to derive that product from the feature model; in other words, variability coverage is the count of number of bounded variation points for a given product. Also, cyclomatic complexity is the number of distinct cycles that can be found in the feature model, which can be shown to be equivalent to the number of integrity constraints on the feature model. The cyclomatic complexity of each product is the number of integrity constraints that are enforced in the product derivation

[2] This is computationally inexpensive using FaMa. `http://www.isa.us.es/fama/`

process of a given product. We define the fitness function ($\mathfrak{F}$) as follows:

$$\mathfrak{F}(c) = \|(\mathcal{VC}(c), \mathcal{CC}(c))\| = \sqrt{\mathcal{VC}(c)^2 + \mathcal{CC}(c)^2}. \quad (1)$$

where c is the chromosome (derived application) for which the fitness is calculated.

Following the *survival of the fittest* strategy, fitter chromosomes survive while the weaker ones are discarded to make room for new chromosomes. We employ the *roulette-wheel selection* method to perform the natural selection process. This method does not necessarily remove all of the weakest chromosomes but associates each chromosome with a removal probability which is inversely proportional to its fitness value. This way, weaker chromosomes also have a chance although slim for survival.

The new chromosomes are generated based on the mutation and cross-over operators. Given the structure of the chromosomes, mutation causes a feature that is not present in the previous chromosome to be included in the configuration or vice-versa (a random gene be flipped in the chromosome); therefore, mutation is quite useful in creating new configurations that are only slightly different from previous configurations (chromosomes), but have the possibility of covering more errors. However, unlike mutation, cross-over is a more radical operator for our case since it rotates the values of the genes around a central pivot gene in two chromosomes. The resulting chromosomes are radically different from the original chromosomes since the values of many genes could possibly change in this process. For our purpose, this operator is useful for avoiding the Genetic Algorithm from getting trapped in a local extremum that could lead to inefficient tests.

It is important to mention that the new chromosomes that are developed based on mutation and cross-over are not necessarily valid feature model configurations. Since we are interested in making sure that the generated tests are valid feature model configurations, the generated chromosomes that are invalid configurations are discarded and are not added as a part of the new generation. This way, the population is always an acceptable test suite.

In terms of the termination conditions for the Genetic Algorithm, two main stoppage criteria were defined: First, a maximum number of generations is defined that ensures that the algorithm will always terminate after a certain amount of time. Second, the algorithm would terminate if the difference between the average fitness values of two generations does not exceed a predefined threshold. While the former condition guarantees the termination of the algorithm, the latter would ensure that the algorithm will terminate once it is not providing any further optimizations. Once the algorithm terminates, the available chromosomes in the population form the generated tests for the given product line. In the following section, we evaluate the quality of the generated tests for several SPLOT feature models in terms of both feature and error coverage.

4 Evaluation

In this section, we will initially introduce the feature models that were employed within our experiments and then will describe the experimental setup and the obtained results.

4.1 Objects of Analysis

For the purpose of our experiments, we have selected a set of feature models that are publicly available through the Software Product Line Online Tools (SPLOT) website [18]. SPLOT's primary 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

Table 1. The feature models used in the experiments and their characteristics.

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
Sienna	38	26%	2,520
Thread	44	0	80,658
Dell Laptop	46	80%	853
HIS	67	11%	6,400

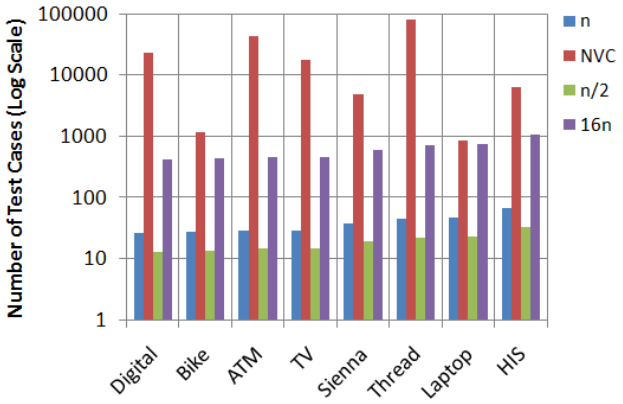

Fig. 3. The number of generated tests in each test suite.

their software product line feature models in an open and free manner. The feature models in this online repository are expressed in SXFM format, which is an XML representation.

The feature models that were used in our experiments along 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 value of zero are those that do not consist of any additional integrity constraints. For the purpose of calculating the NVC of each feature model, the binary decision diagram technique proposed by Mendoca et al. [17] was employed. We also benefited from the FaMa toolkit for validating the feature model configurations. FaMa is a useful framework for the automated analysis of feature models. It consists of some of the most commonly used logical representations and configuration techniques for feature models.

4.2 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 convert them to the required Binary chromosome format to be used with the Genetic Algorithm implementation provided in JGAP [16]. The program will create the representation of the software

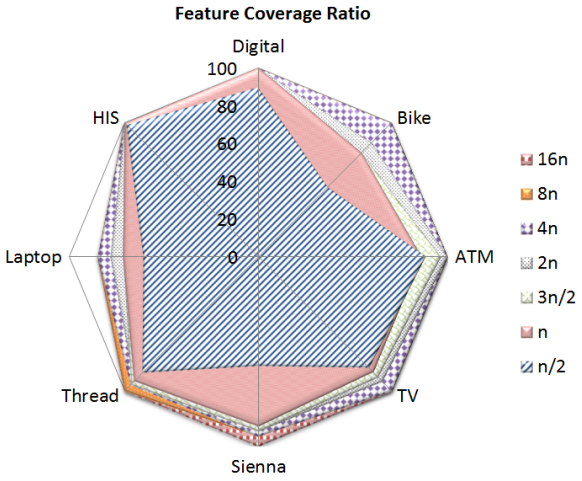

Fig. 4. Feature coverage based on test suite size.

product line feature model configurations as chromosomes. This program also interacts with the FaMa toolkit to validate each of the chromosomes that are developed in the Genetic Algorithm. This interaction will ensure that all of the chromosomes that are available in the Genetic Algorithm population are valid feature model configurations and are hence appropriate tests for the product line.

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 efficiency of our coverage criteria and their corresponding test suite generators, we developed an *error generation simulator*. The lack of publicly available datasets has already been pointed out in [8, 2] and several authors and tool suites (such as FaMa [3] and 3-CNF model generator [18]) have already 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 errors for the feature model correspondingly. Feature models with more features, higher ratio of CTCR and a higher number of valid configurations will contain more errors generated by the simulator. The generated errors are in one of the following categories:

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

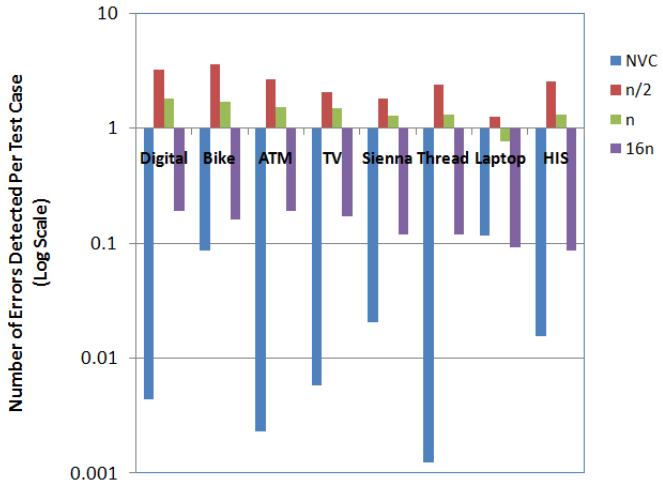

Fig. 5. Error detected per test.

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

Further details and code for the error generator is available for download at `http://ebagheri.athabascau.ca/splt/splt.zip`. Interested researchers are encouraged to use it for replication studies. The error generation simulator will create a set of errors for each feature model in the form of the above three error categories, which are then used to test how well the test suite generation technique is able to detect the generated errors for each of the feature models. The results of the evaluation based on the generated errors are reported in the following.

4.3 Results and Analysis

Our experiments were performed on a dedicated 2.8GHz Pentium IV Personal Computer with 2GB of memory, running a Windows XP operating system along with Java 6.0. In order to be able to study the tradeoff between feature and error coverage of our approach, we have performed multiple experiments in each of which we have selected a different population size for the Genetic Algorithm. As discussed earlier, since the population size of the Genetic Algorithm represents the size of the generated test suite, we are interested to see if and to what extent the size of the test suite impacts error coverage. For each of the feature models shown in Table 1, we have generated test suites based on the Genetic Algorithm with population sizes of (n being the number of features in the feature model): $\frac{n}{2}$, n, $\frac{3n}{2}$, $2n$, $4n$, $8n$, and $16n$. This means that, e.g., for the Thread feature model, we generate test suites of sizes: 22, 44, 66, 88, 176, 352, and 704. As seen, the largest generated test suite only contains 704 tests, whereas if the total number of possible tests were to be tested, 80,658 configurations would have been needed to be considered. Figure 3 shows a comparison of the number of tests in each test suite generated by our approach with the

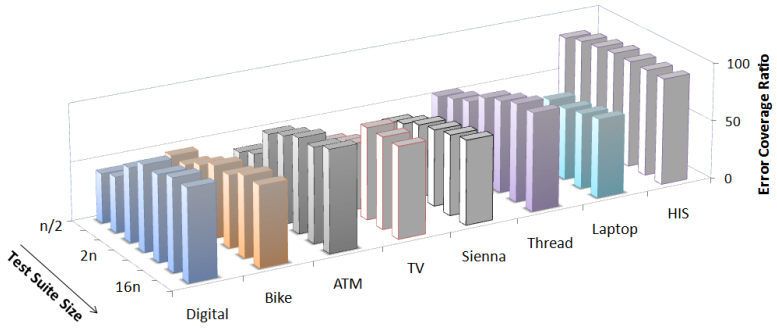

Fig. 6. Error coverage by different test suite sizes.

possible set of tests for the feature model (NVC) in log scale. It can be seen that even for the test suites with $16n$ tests, the size is smaller than the test suites generated by NVC.

Given the reduction in the size of the test suite, the important questions that need to be studied are: Q1) does the reduced test suite size provide sufficient feature coverage, i.e., do the test suites provide tests that test each feature of the feature model at least once? Q2) how would a smaller test suite impact error coverage, i.e., do smaller test suites lack the required precision to identify the possible errors in a software product line?

With regards to Q1, it is clear that test suites based on NVC have a complete feature coverage. In other words, it is guaranteed that they will test all of the features of the feature model. However, in our approach since the size of the test suites are much smaller, full feature coverage is not always ensured. As the size of the test suite grows larger, higher feature coverage is achieved, which is shown in Figure 4. Despite the fact that test suite size impacts feature coverage, an important observation can be made based on Figure 4, which is that the increase of test suite size after a certain amount does not significantly impact feature coverage. As seen in the figure, for the feature models in our experiments, a test suite size of $4n$ has been able to provide good feature coverage and larger test suite sizes have not been able to significantly improve this feature coverage compared to $4n$. So, as a matter of tradeoff between test suite size and feature coverage, one could argue that the advantages of a four times smaller test suite ($4n$ vs $16n$) in terms of time and cost of performing tests outweighs a $4-5\%$ increase in feature coverage. Yet, the decision about this tradeoff is dependent on the domain being tested and the test engineers preferences.

Now regarding the impact of test suite size on error coverage (Q2), it is important to first study how efficient each of the tests of a test suite are. For this purpose, we compute the *average number of errors detected per test* in a generated test suite. The results are shown in Figure 5 in log scale. It is interesting to observe that as the size of the test suites increases, the effectiveness of the tests in identifying errors decreases significantly. This indicates that as the size of the test suite increases and more tests are considered, only marginal improvements in terms of error coverage is achieved. To put this observation in context, we would need to examine the trend of error coverage improvement given different test suite sizes. Figure 6 depicts this trend for the different objects of analysis and for different test suite sizes. It can be seen in this figure that the increase in error coverage occurs up until test suites with a size of $2n-4n$, and further increase in test suite size only has a minor impact on the increase of error coverage. It seems that increasing the size of test suites is not necessarily a better strategy for testing software product lines, since although larger test

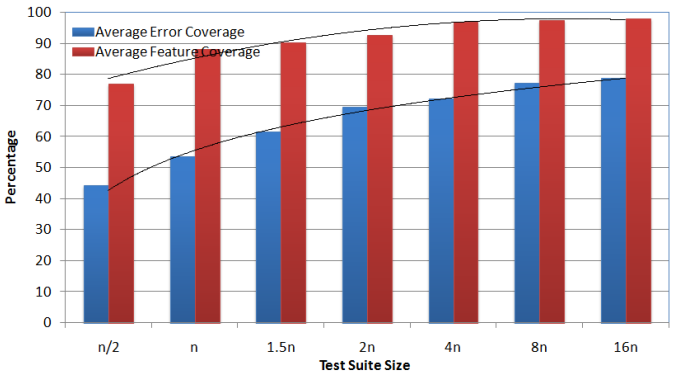

Fig. 7. Impact of test suite size on error and feature coverage.

suites increase the cost associated with testing a product line, they only provide marginal benefits in terms of feature and error coverage.

Figure 7 shows the average feature and error coverage over all objects of study for different test suite sizes. Besides showing the fact that increasing the size of the test suite beyond a certain point is not necessarily a desirable choice, this figure also exposes an implicit relationship between feature coverage and error coverage, i.e., a test suite with a higher feature coverage is more likely to have a higher error coverage. This is further shown in Figure 8 where each of the points in the figure is a representative of a test suite generated for one of the objects of analysis with a defined test suite size. Although this correlation between feature and error coverage is expected, the degree of impact of feature coverage on error coverage is attractive in that it shows that even test suites with complete feature coverage (100%) do not necessarily identify all of the errors in a feature model. This can be explained as follows: According to Section 4.2, errors can be in three different classes. A complete feature coverage will only ensure that *Errors in individual features* are covered. The other two classes are not guaranteed to be fully covered unless test suites based on NVC are comprehensively generated and tested, which is both cost and computational wise infeasible. Therefore, reaching a tradeoff between the size of the test suite and the error and feature coverage as achieved in Figures 6 and 7 seems to be a viable strategy.

We believe that the our approach for generating test suites has been successful in terms of its ability to cover both existing errors and features given small test suite sizes. As an example, our approach has been able to reach ˜80% error coverage (Figure 6) and ˜90% feature coverage (Figure 4) with only a test suite of size $4n = 704$ tests for the Thread feature model. This can be considered a rather efficient tradeoff if we consider the fact that a 100% feature and error coverage would only be achieved if a test suite of size $80,658$ tests is evaluated, which requires much higher testing costs and resources.

5 Related Work

As software product lines become more popular for facilitating reuse and enhancing efficiency and production time [22], issues related to quality engineering become of more importance. Some authors already believe that software product line development processes still lack appropriate testing techniques that can address commonality and variability inherent in the product lines [23, 19]. Although generic software testing approaches can be

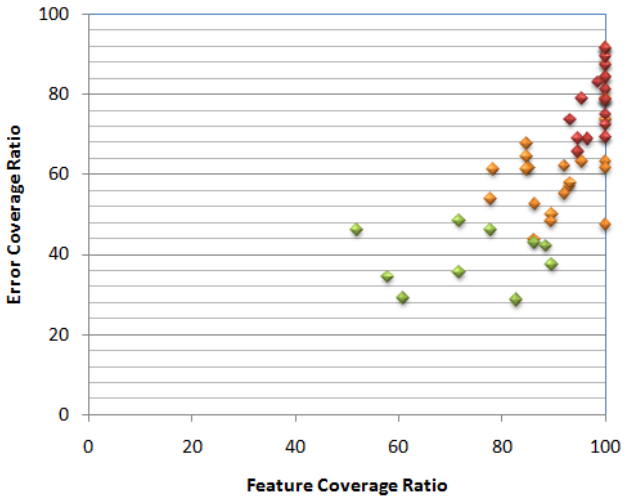

Fig. 8. Correlation between error and feature coverage (hotter colors show more desirable tests).

used on products of the product line, an entire product line is much harder to be tested using these techniques due to the numerosity of possible configurations of the models and the complex feature interactions that may exist [24]. Many of the techniques for software product line testing have focused on ensuring quality within the process of software product line development and planning [9, 20, 7]. However, the focus of our work in on the development of appropriate test suite generation strategies for product line feature models.

There are only a limited number of approaches that directly address the issue of test generation in software product lines. Our approach is significantly different from the related work in that we employ an evolutionary approach for searching the test space, which has not been explored in the area of software product lines. Our approach is mainly guided by a fitness function that centers on variability and cyclomatic complexity of the feature model. The main benefits of our approach compared to other approaches can be seen as being very computationally efficient as it does not perform complex optimization operations, provides a tradeoff between feature and error coverage and develops tests that focus on specifically evaluating variability and integrity constraints.

In contrast, some related techqniques employ automatic analysis based on SAT solvers [12] such as Alloy. For instance, Uzuncaova et al. propose a hybrid approach by combining methods from software product lines and specification-based testing using Alloy [29]. In their approach, each product is defined as a composition of features represented as Alloy formulae. The Alloy analyzer is then used to incrementally generate tests over partial specifications of the products. This approach is an improvement over previous work that generated tests in a single pass execution of Alloy over complete specifications [28].

In a different work, Perrouin et al. employ T*-wise test generation* [21]. 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. For this, they 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 [13] base their work on the assumption that although the number of product line configurations are exponential in the number of available features but the 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 tests 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 and hence reduce the size of the test space. Other work which reduce the test space mainly focus on the use of user requirements to identify the most important set of features that need to be tested [25]. In such approaches, it is the end users' needs that drive the test generation process and not the feature interactions.

Less relevant to our line of work in this paper is the proposal by Segura et al [26]. In their paper, the authors develop a set of *metamorphic* relations between the input feature models and their product and use this as a way to generate appropriate tests. However, the tests generated in this approach are formed in such a way to test automated feature model configuration programs and not software product line feature models.

6 Concluding Remarks

In this paper, we have proposed a search-based testing approach based on the evolutionary Genetic Algorithms to automatically generate test suites for software product lines. The main issue with testing product lines is the exponential number of configurations that need to be tested. We have exploited the exploration capabilities of Genetic Algorithms to search the *configuration space* of a feature model and to find the subset that provides suitable error and feature coverage. We have performed experiments using some models provided by the SPLOT repository and have shown that our proposed test suite generation approach is able to develop test suites that provide a reasonable tradeoff balance between error and feature coverage. This is achieved by only generating test suites with a size complexity of $O(n)$ as opposed to $O(2^n)$. We are currently evaluating our test suite generation approach on larger synthetic feature models that are generated by the SPLOT and FaMa toolkits. The results of these experiments can further show the performance of the generated test suites with regards to error and feature coverage. Also we are interested in comparing our test generation technique with some other similar approaches in the literature. Currently, an implementation of the proposed work by other researchers is not publicly available. We are releasing the implementation of our work at `http://ebagheri.athabascau.ca/splt/gasplt.zip` in hopes of future comparative studies.

References

1. Bagheri, E., Deldari, H.: Dejong function optimization by means of a parallel approach to fuzzified genetic algorithm. Computers and Communications, IEEE Symposium on 0, 675–680 (2006)
2. Bagheri, E., Gasevic, D.: Assessing the maintainability of software product line feature models using structural metrics. Software Quality Journal 19(3), 579–612 (2011)
3. Benavides, D., Segura, S., Trinidad, P., Ruiz-Cortes, A.: 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). pp. 129–134 (2007)
4. Buehler, O., Wegener, J.: Evolutionary functional testing of an automated parking system. In: International Conference on Computer, Communication and Control Technologies (CCCT'03). Citeseer (2003)
5. Clements, P., Northrop, L.M.: Software product lines, visited june 2009, http://www.sei.cmu.edu/programs/pls/sw-product-lines_05_03.pdf (2003)
6. Coello, C., Lamont, G., Van Veldhuizen, D.: Evolutionary algorithms for solving multi-objective problems. Springer-Verlag New York Inc (2007)

7. Cohen, M., Dwyer, M., Shi, J.: Coverage and adequacy in software product line testing. In: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis. p. 63. ACM (2006)
8. Cohen, M.B., Dwyer, M.B., Shi, J.: Coverage and adequacy in software product line testing. In: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis. pp. 53–63. ROSATEA '06, ACM, New York, NY, USA (2006)
9. Dallal, J., Sorenson, P.: Testing software assets of framework-based product families during application engineering stage. Journal of Software 3(5), 11 (2008)
10. Goldberg, D.: Genetic Algorithms in Search and Optimization (1989)
11. Kang, K., Lee, J., Donohoe, P.: Feature-oriented product line engineering. IEEE software 19(4), 58–65 (2002)
12. Khurshid, S., Marinov, D.: TestEra: Specification-based testing of Java programs using SAT. Automated Software Engineering 11(4), 403–434 (2004)
13. Kim, C., Batory, D., Khurshid, S.: Reducing Combinatorics in Testing Product Lines (Technical Report), 2010, UTexas)
14. Lee, K., Kang, K., Lee, J.: Concepts and guidelines of feature modeling for product line software engineering. Lecture Notes in Computer Science 2319, 62–77 (2002)
15. McMinn, P.: Search-based software test data generation: A survey. Software Testing, Verification and Reliability 14(2), 105–156 (2004)
16. Meffert, K., Rotstan, N., Knowles, C., Sangiorgi, U.: JGAP–Java Genetic Algorithms and Genetic Programming Package. URL: http://jgap. sf. net
17. Mendonca, M., Wasowski, A., Czarnecki, K., Cowan, D.: Efficient compilation techniques for large scale feature models. In: Proceedings of the 7th international conference on Generative programming and component engineering. pp. 13–22. ACM New York, NY, USA (2008)
18. Mendonca, M., Branco, M., Cowan, D.: 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. pp. 761–762 (2009)
19. Nebut, C., Fleurey, F., Le Traon, Y., Jzquel, J.M.: A requirement-based approach to test product families. In: van der Linden, F. (ed.) Software Product-Family Engineering. Lecture Notes in Computer Science, vol. 3014, pp. 198–210. Springer Berlin / Heidelberg (2004), http://dx.doi.org/10.1007/978-3-540-24667-1_15, 10.1007/978-3-540-24667-1_15
20. Olimpiew, E., Gomaa, H.: Model-based testing for applications derived from software product lines. In: Proceedings of the 1st international workshop on Advances in model-based testing. p. 7. ACM (2005)
21. Perrouin, G., Sen, S., Klein, J., Baudry, B., Le Traon, Y.: Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In: ICST 2010. pp. 459–468 (2010)
22. Pohl, K., Böckle, G., Van Der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer (2005)
23. Pohl, K., Metzger, A.: Software product line testing. Communications of the ACM 49(12), 81 (2006)
24. Reis, S., Metzger, A., Pohl, K.: Integration testing in software product line engineering: A model-based technique. Fundamental Approaches to Software Engineering pp. 321–335 (2007)
25. Scheidemann, K.: Optimizing the selection of representative configurations in verification of evolving product lines of distributed embedded systems (SPLC 2006)
26. Segura, S., Hierons, R., Benavides, D., Ruiz-Cortés, A.: Automated test data generation on the analyses of feature models: A metamorphic testing approach. In: ICST 2010. pp. 35–44 (2010)
27. Tracey, N., Clark, J., Mander, K.: Automated program flaw finding using simulated annealing. In: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis. p. 81. ACM (1998)
28. Uzuncaova, E., Garcia, D., Khurshid, S., Batory, D.S.: A specification-based approach to testing software product lines. In: ESEC/SIGSOFT FSE. pp. 525–528 (2007)
29. Uzuncaova, E., Khurshid, S., Batory, D.S.: Incremental test generation for software product lines. IEEE Trans. Software Eng. 36(3), 309–322 (2010)
30. Watkins, A.: The automatic generation of test data using genetic algorithms. In: Proceedings of the 4th Software Quality Conference. vol. 2, pp. 300–309 (1995)