Nr.: FIN-03-2014

Scalable and Efficient Sampling for Product-Line Testing

Mustafa Al-Hajjaji

Arbeitsgruppe Datenbanken und Software Engineering

Nr.: FIN-03-2014

Scalable and Efficient Sampling for Product-Line Testing

Mustafa Al-Hajjaji

Arbeitsgruppe Datenbanken und Software Engineering

Technical report (Internet)
Elektronische Zeitschriftenreihe
der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg
ISSN 1869-5078

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 5 TMG)

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Mustafa Al-Hajjaji
Postfach 4120
39016 Magdeburg
E-Mail: m.alhajjaji@iti.cs.uni-magdeburg.de

http://www.cs.uni-magdeburg.de/Technical_reports.html
Technical report (Internet)
ISSN 1869-5078

Redaktionsschluss: 15.10.2014

Bezug: Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Dekanat

Scalable and Efficient Sampling for Product-Line Testing

Mustafa Al-Hajjaji
University of Magdeburg
Magdeburg, Germany

ABSTRACT

A software product line (SPL) is a family of software products that share a set of common features. Exhaustively testing every product of an SPL is a difficult task due to the combinatorial explosion of the number of products. Several sampling approaches have been proposed to select a set of products that can be used to test SPL. However, these approaches do not scale very well, especially for large SPLs such as the Linux Kernel. In addition, it is typically up-to testers in which order these products are tested. The testers may wish to order these products to detect faults as soon as possible. The products have been prioritized based on domain knowledge or feature model criteria, but not much attention has been paid to criteria at code-based level. We plan to use evolutionary testing approaches with different inputs to fitness functions to explore the configuration space of SPL feature models. Using the criteria from feature-model and code as inputs to evolutionary testing approaches, we want to investigate whether we can increase the efficiency of SPL testing w.r.t. finding more faults. Furthermore, we want to investigate code-based metrics which can be used to enhance SPL testing. In this proposal, we present research questions, research methods and a concrete working plan to investigate how we can employ evolutionary testing approaches to increase the efficiency of SPL testing.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Testing—*Reusable Software*; D.2.13 [**Software Engineering**]: Reusable Software—*domain engineering*

General Terms

Reliability, Verification

Keywords

Software product-line testing, software testing, sampling, evolutionary testing

1. INTRODUCTION

SPL engineering is an approach reusing a common set of features across a very large number of similar products systematically. Recently, SPLs are gaining acceptance and several companies such as Boeing, Bosch, Hewlett Packard, Toshiba, and General Motors adopt their software development process to SPLs [2]. SPL engineering addresses well-known needs of software engineering, such as reducing the cost of development and maintenance, increasing the quality, and decreasing the time to market [30].

The commonality among the products in SPLs represents very large portion of the functionality of those products, it gives opportunity to reduce the resources required in all development phases including testing. The variation of SPLs adds new dimensions of complexity to traditional testing. Testing an SPL is a difficult task due to the amount of possible combinations between features which lead to explosion of possible products that need to be tested. The main concern is that it is not possible to test all the possible products because the resources for testing are usually limited. To tackle this problem, several approaches have been proposed to reduce the number of products to test [7], such as combinatorial interaction testing (CIT).

CIT is a promising approach that can perform the interaction testing between features in an SPL [11]. CIT has been proposed to sample a set of configurations to find interactions faults [10]. This approach is based on the observation that most of the faults are a result of an interaction of few features. Pairwise interaction testing is one of the techniques that has been used to achieve CIT [32] [31]. Pairwise testing is used to detect faults that are caused by the interaction between two features. Recently, T-wise testing has been generalized from pairwise testing to cover all t-wise combinations between features in SPL testing [22].

The challenge is that computing all the t-wise interaction from a feature model with the presence of all the constraints is still a problem, since the existing sampling approaches do not scale well, especially for a large feature model such as Linux kernel (over 11,000 features [13]) [22] [26].

Although CIT is a technique to reduce the number of products to test, the number of the products can be too large to test especially if the time budget is limited. For example, 480 products are required to be tested for part of Linux kernel (only 6888 features) with 2-wise coverage interaction [22]. Hence, some approaches are required to prioritize the products to start testing with the most relevant products. The idea of prioritizing products is to find faults as soon as possible by starting testing the products that are most likely to contain faults.

The existing prioritization approaches consider only the domain knowledge [4] [15] and feature model [1] [34]. To the best of our knowledge, there is no any approach considers code-based criteria in SPL testing.

Throughout this paper, we will use the term *efficiency* to represent the following: finding more faults and increasing the rate of fault detection (i.e. a measure of how quickly

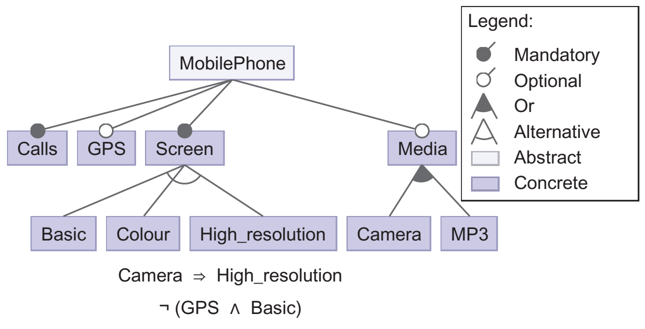


Figure 1: Feature Diagram of *Mobile Phone* SPL

faults are detected). It is better for testers to detect the first fault after half an hour of testing than waiting five hours.

There are a few approaches have proposed to combine sampling and prioritization [19] [16]. Ensan et al. [16] and Henard et al. [19] use genetic algorithms to generate products based on certain objectives. In their approaches, they focus only on a few criteria based on feature models.

We plan to propose evolutionary algorithms to explore the configuration space of feature models in order to generate the optimal products based on different objectives. We want to investigate whether the cuckoo search (CS) algorithm [40] can be used to solve the identified problems. Yang and Deb [40] have shown that CS is superior to existing evolutionary algorithms (genetic algorithms and particle swarm optimization) for multimodal objective functions. In addition, we want to investigate whether the code-based criteria can be used as input to the fitness functions of CS to enhance the efficiency of SPL testing. In addition, we plan to include the domain knowledge, when it is available, specifications, and implementation as inputs to the fitness functions of evolutionary algorithms.

2. FEATURE MODELING

A configuration is a valid combination of features in an SPL with respect to the feature model [24]. Each configuration can be used to generate a product. A feature model is used to define the valid combination of features [24]. A feature model can be represented graphically by feature diagram structured as a tree.

In a feature diagram, several types of connections between the features and the subfeatures can be represented as illustrated in *Figure 1*. These connections are distinguished as: *or* and *alternative* groups [5]. Subfeatures can be *optional*, such as *GPS* and *Media*, or *mandatory*, such as *Calls* and *Screen*. In an *or*-group, at least one subfeature has to be selected, when its parent is selected. In an *alternative*-group, exactly one subfeature must be selected [36], when its parent is selected. For instance, a mobile phone can include only one of the following features: *Basic, Colour*, or *High_resolution*. In addition, the selection of a feature implies the selection of its parent feature and its mandatory subfeatures.

The features are either *abstract* or *concrete:* it is *concrete*, if implementation artifacts are mapped to that feature; otherwise, it is *abstract* [37]. Furthermore, features can have additional dependencies that cannot be described using only the hierarchical structure; cross-tree constraints are used to define such dependencies. Cross-tree constraints are propositional formulas usually shown below the feature diagram [2]. As illustrated in *Figure 1*, the feature model of the *Mobile Phone* SPL contains cross-tree constraints *requires* and *excludes*. An example of a *requires* constraint is, if feature *camera*is selected to be included in a mobile phone, the feature *High_resolution* of the screen must be selected. An example of an *excludes* constraint is that a mobile phone cannot support the features *GPS* and *Basic* at the same time.

3. STATE OF THE ART

We divide this section into three parts. In the first part, we present prioritization techniques in SPL testing. We present the sampling techniques that use constraint-based approaches in the second part. In the third part, we discuss the search-based approaches that are used to sample the configurations.

Product Prioritization.

Concerning prioritizing the products, a few approaches have been proposed to prioritize them based on different criteria. Baller et al. propose a framework to select adequate test suites for sets of software variants under test with regard to *cost* of test cases and *profit* of covering test requirements [4]. They do not sample the products; the products are given as input to their approach. Devroey et al. [12] propose an approach to prioritize products based on the behavior of SPL. They use statistical testing to extract products with high probability to be executed. Ensan et al. propose a goal-oriented approach by giving a high priority to the products that contain the most desirable features [15]. Ensan et al. [15] prioritize these products depend on the expectations of the domain stakeholders. The approaches of Devroey et al. [12] and Ensan et al. [15] need domain knowledge to prioritize the products.

Constraint-Based Sampling.

Several approaches have been proposed to reduce the number of products under test. *Combinatorial interaction testing (CIT)* is a strategy used to select a subset of products in which interaction faults are most probably to occur. Several approaches use t-wise sampling to achieve the combination interaction between features [9, 10, 32]. Oster et al. [31] introduce a methodology to apply pairwise testing approach to a feature model by combining graph transformation and forward checking. Oster et al. [31] ensure that preselected products are part of products that are generated from pairwise coverage. Perrouin et al. [33] transform the feature model into Alloy to select configurations, then they use a SAT solver to check the validity of these configurations. This approach faces scalability issues due to the additionally step of transforming the Alloy model to conjunctive normal form (CNF) before it can be checked by a SAT solver [20]. The approaches of Oster et al. [31] and Perrouin et al. [33] do not consider any type of prioritization for the generated configurations. Johansen et al. combine the sampling and prioritization [23]. In their approach, they use domain knowledge about which feature interactions are prevalent in the market to assign weights to sub-product lines. Uzuncaova et al. propose an approach by generating test cases from software product line specifications [38]. The challenge in these approaches is that it is not easy to have the domain knowledge of the SPL.

Selecting which test cases need to be run is another strategy used to reduce the test space. Kim et al. use static analysis to reduce the number of configurations by considering features as combinatorial parameters [25]. They use the static analysis of test cases and source code to reduce the number of variant that need to be tested. Shi et al. present a compositional symbolic execution technique to analyze an SPL [35]. They use the interactions between features to reduce the number of products that need to be tested. Both approaches of Kim et al. [25] and Shi et al. [35] reduce the test space of an SPL by generating a subset of all the possible configurations, but they do not prioritize these configurations to increase the efficiency of SPL testing.

Search-Based Sampling.

To the best of our knowledge, there are only two search-based approaches to generate a subset of all possible configurations in SPL testing. Ensan et al. [16] use genetic algorithms to generate these configurations with a high fault-detection capability. Henard et al. [19] use genetic algorithms to generate the subset configurations using t-wise covering and prioritize them based similarity heuristics. In addition, they use different criteria and they represent the gene in genetic algorithm in a different way. Ensan et al. [16] consider each product is a gene and each feature is a chromosome, while Henard et al. [19] consider the set of products as a gene and each product is a chromosome. Both of these approaches have been proposed to generate and select the configurations to be tested, but the input parameters to the fitness function of their approaches are defined only at the feature-model level and they do neither incorporate source code nor test cases.

4. PROBLEM STATEMENT

The high number of possible combinations of features which often leads to a combinatorial explosion of possible products makes testing an SPL a difficult task. Since the resources in testing are usually limited, it is not possible to test all the possible products. Until now, there is no consensus on how to efficiently test SPLs [14]. Most of current sampling algorithms apply t-wise testing for t<=3 [22]. Kuhn et al. mention that 70 % and 95 %, of faults are found for 2-wise and 3-wise coverage respectively [27]. They mentioned that the larger interaction coverage between features achieved, the higher percentage of faults are found. They indicate that approximately 100 %, of faults are found for 6-wise coverage. Thus, to detect almost all faults, we need to apply 6-wise interaction testing. The challenge of applying 6-wise testing is that the number of products is still high especially for large feature models. For example, 480 products are required to be tested for Linux kernel(with 6888 features) with only 2-wise coverage interaction [22]. Applying 6-wise testing will increase the number of products significantly. Several sampling approaches have been proposed to select a set of configurations to be tested as representative of an SPL [31] [33] [21]. Although these approaches have been improved recently [22], but they still have a scalability problem of handling large feature models and high interaction coverage between features [26]. In addition, theses sampling approaches focus only on how to generate the minimum number of products for a given coverage. However, there are other parameters that may involved in the SPL testing process, such as time, profit, and cost [4, 4, 19].

The time budget of testing is usually limited. Hence, in which order products are tested is important in this context. The testers wish to order the products in a way to help them to detect the faults as soon as possible. The tester can save time by starting testing the products that are most likely contain faults. The main goal of prioritizing products is to increase the efficiency of SPL testing w.r.t. increasing the rate of fault detection.

Several criteria are investigated whether they can enhance the process of selecting and prioritizing products. The existing approaches have focused on domain knowledge and feature-model criteria. To the best of our knowledge, there is no work consider code-based criteria in prioritizing products.

We plan to use an evolutionary testing approach to explore the test space of SPL and select a subset of the possible configurations. In addition, we are going to prioritize the selected configurations based on different criteria from feature-model level and code-based level. In addition, we will consider specification and the domain knowledge, when it is available, during the prioritization process. In particular, we address the following research questions:

- **RQ1:** How to increase the rate of fault detection for SPLs testing by prioritizing products?
- **RQ2:** How to achieve efficient sampling (i.e. handling multiple parameters)?
- **RQ3:** What are code-based metrics that enhance the efficiency of SPL testing?

5. RESEARCH APPROACH

To answer RQ1, we propose a similarity-based prioritization approach to prioritize the configurations [1]. Our initial results indicate that the order of our approach is better than the random orders and is often better than the default order of sampling algorithms which are used to sample configurations from feature models.

To answer RQ2, we plan to employee metaheuristic search algorithms, which are general algorithmic frameworks designed to solve complex optimization problems [6], to search the possible configuration space of feature models (see Section 5.1). The key to answer QR2 is the fitness function. It is used to guide Metaheuristic algorithms. The main role of the fitness function is to capture SPL testing parameters (objectives). The Metaheuristic algorithms seek inputs to the fitness function that maximize the efficiency of SPL testing. Metaheuristic algorithms are very generic, because different fitness functions can be defined to achieve different goals. Hence, we can apply the same Metaheuristic algorithm to very different SPL testing scenarios. In Section 5.2, we investigate how we can address RQ3.

5.1 Metaheuristic Search

The test case generation process in the industrial sector is often part of test engineer responsibility. This process takes a long time and consumes a plenty of effort, especially for large applications. Hence, many test engineers favor metaheuristic search techniques to generate appropriate tests automatically. If an optimal solution is required for

Algorithm 1: Pseudocode of cuckoo search [40]

```
1: Objective function $f(x)$, $x=(x_1, x_2, ..., x_d)$;
2: Generate initial population of
   $n$ host nests $x_i (i=1,2,...,n)$;
3: while ((t <MaxGeneration) or stop criterion)
4:     Get a Cuckoo randomly by Levy flights;
5:     Evaluate its quality/ Fitness $F_i$;
6:     Choose a nest among $n$ (say j) randomly;
7:     if $F_i \geqslant F_j$
8:         replace $j$ by the new solution;
9:     end
10:      Abandon a fraction (pa) of worse nests
         [and build new ones via Levy flights]
11:      Keep the best solutions
         (or nest with quality solutions);
12:      Rank the solution and find the current best;
13: end while
14: Post process results and visualization;
```

combinatorial problems at reasonable computational cost, metaheuristic search techniques can be used to achieve this task [29].

Several metaheuristic techniques, such as hill climbing, simulated annealing, and evolutionary algorithms have been used in the software test generation [8] [41] [39]. Hill climbing is a local search algorithm used to improve one solution. This solution is randomly chosen from the search space at starting point, then attempts to find a better solution by incrementally changing a single solution at a time. If a better solution is found, then this replaces the current solution. The previous step will be repeated until no improved solutions can be found for the current solution [29]. Hill climbing is simple and gives a fast result. Simulated annealing is similar to hill climbing, but it allows more freedom movement around the search space. Evolutionary algorithms use simulated evolution as a search strategy to evolve candidate solutions [29]. Evolutionary testing refers to the process of using evolutionary algorithms to generate test cases [29]. In our project, at first, we plan to exploit the evolutionary algorithms using cuckoo search (CS) algorithm [40] to generate products for software product line. Then, we will compare CS against well-known evolutionary algorithm (e.g., genetic algorithm) [16] [19].

Cuckoo Search Algorithm.

The cuckoo search (CS) optimization algorithm is a recently proposed metaheuristic, based on the brood parasitism behavior of certain species of the cuckoo birds such as the *guira* and the *ani*, combined with the displacement mode of many animals and insects such as the sharks and the some bird species. This movement pattern has been described as a heavy-tailed probability distribution and called the Lévy Flight pattern [40].

In order to simulate the behavior of the cuckoo birds, the original authors adopted three idealizing rules which are as follows:

- Each cuckoo lays one egg at a time. The cuckoo chose a random nest to dumb its egg.
- The next generations will be carried by the best nest with high quality of eggs.
- The number of available host nests is fixed and a host can discover an alien egg with a probability [0; 1].

If the host discovers an alien egg, the host bird can either throw the egg away or abandon the nests and build a new nest in a new location via lévy flight algorithm. Lévy flight algorithm helps to efficiently explore the neighborhood of the current best obtained a set of configurations in terms of measures (related to the inputs of fitness function) without getting trapped in local minima [40].

Algorithm 1 summarizes the main functioning scheme of the CS. The algorithm starts by initializing a random population of size n from the search space, sometimes also called the belief space. This initial population is sorted and ranked based on a fitness function, and a new candidate solution is generated starting from the best so far obtained solution in the initial population. This new candidate is generated by performing a Lévy Flight around the previous best solution. Later, a randomly chosen solution from the initial population is compared to the newly generated Lévy candidate in terms of their fitness function. If the new element is of a better quality, the randomly chosen element is substituted. The discovery probability process described in *Algorithm* 1 is represented by a substation process in the initial population. This means, a portion of the worst nests from the so far obtained population is replaced by randomly generated one with a probability p_a. The last obtained population is ranked and the best element is evaluated against the stop criteria. This process describes the basic operations of each iteration, and the algorithm performs as many iterations as necessary until one of the stop criteria is met (e.g., time).

In order to properly project this algorithm on the SPL problems, each solution would represent a configuration. Figure 2 shows one of the many possible applicable diagrams to solve our problem using the CS algorithm.

Genetic Algorithm.

Genetic algorithms are inspired by the processes of Darwinian evolution. Genetic algorithm is one of the evolutionary algorithms which are used to select superior candidate solutions. The first step in genetic algorithms is similar to the one in CS which is generating a set of configurations randomly. These configurations serve as the initial population for genetic algorithm. In the second step, each configuration is evaluated using fitness function. The weaker configurations w.r.t. fitness function are discarded in the third step. In the fourth step, the rest of the configurations are used as seeds for crossover and mutation operators to generate the next generation of configurations. The new configurations are evaluated (step 2). The process will continue until the condition (e.g., time) has been satisfied [16].

Genetic algorithms already have been proposed to sample configurations in an SPL [16] [19]. The major difference between both approaches is the way they represent the gene in genetic algorithm. To evaluate our approaches, we plan to conduct comparisons between CS with each one of these approaches.

5.2 Prioritization Criteria for SPLs

Regarding RQ3, we will present examples of feature model and code-based criteria. We plan to investigate whether these criteria and others can increase the efficiency of SPL testing. We will use the following criteria as inputs to the fitness function to prioritize the configurations:

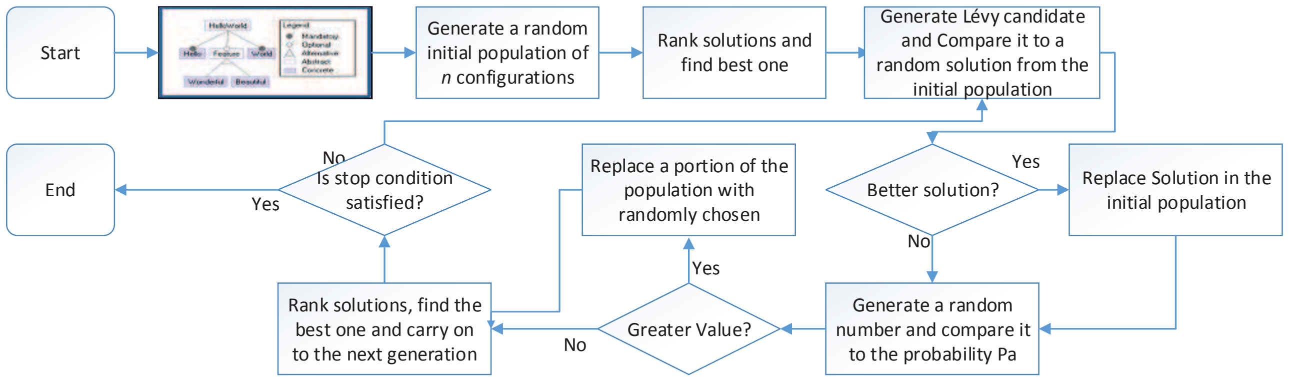

Start
Generate a random initial population of *n* configurations
Rank solutions and find best one
Generate Lévy candidate and Compare it to a random solution from the initial population
Better solution?
Yes
Replace Solution in the initial population
No
Generate a random number and compare it to the probability Pa
Greater Value?
Yes
Replace a portion of the population with randomly chosen
No
Rank solutions, find the best one and carry on to the next generation
Is stop condition satisfied?
No
Yes
End

Figure 2: The overview of multi-objective sampling approach

The similarity between configurations.

This criterion measures the similarity between configurations [1]. At first, we select a configuration, then we calculate the similarity between configurations. We hope that the configuration that has the minimum similarity with all previous tested configurations is the best configuration. The motivation of using the similarity between configurations is that similar products are likely to contain the same faults. Hemmati et al. [18] observe that dissimilar test cases are likely to detect more faults than the similar ones. The similarity values between configurations are between 0 and 1. If the value is 1, it indicates that the configurations are identical. If the value is 0, it means that the configurations are completely different from each other.

We evaluate the similarity between configurations using the Hamming distance [17]. Hamming distance is mathematically given by the following formula, where c_i and c_j are configurations and F is the set of all features in a SPL.

$$d(c_i, c_j, F) = 1 - \frac{|c_i \cap c_j| + |(F \backslash c_i) \cap (F \backslash c_j)|}{|F|} \qquad (1)$$

As it is illustrated in *Equation 1*, we consider not only the selected features when we calculate the distance between configurations, but also the deselected ones. In the first step, we select the first the configuration with the maximum number of selected features because it covers most faults in individual features and enables selection of the next configuration with large distance. It is common in the Linux community to test the configuration with the maximum number of selected features (a.k.a. *allyesconfig*) [13]. The second step is to calculate the distance between this configuration and the other configurations and we select the one with the maximum distance. We continue the process until all the configurations are ordered.

Cross-Tree-Constraints Ratio (CTCR).

This metric measures the degree of involvement of features in the integrity constraints. The input to this metric is a feature model and the output is the ratio of the number of features in the cross-tree constraints to the total number of features in the model [3]. Since we are interested in CTCR for a specific product, CTCR can be defined as follows:

$$CTCR\ (p,fm) = \frac{\#constraintsfeatures(p, fm)}{\#features(p)} \qquad (2)$$

Where *#constraintsfeatures(p,fm)* is the number of features in p involved in constraints and *#features(p)* is the total number of features in product p. The previous metrics are feature-model-based level, the following metric is an example of code-based level metrics that can be used as input in the fitness function. We assume that this criterion can reduce testing effort by starting testing the more complex products in terms of constraints.

Number of Features Constants (NOFC).

The NOFC metric provides perceptions into the variability and complexity of the SPL, since it represents the configuration dimension of an SPL. NOFC metric is measured by calculating how many times each feature occurs in the expressions and summing them per product [28]. According to this criterion, the products that have the maximum number of features will be given priority to be tested.

Lines of Feature Code (LOF).

LOC metric is a code-based metric. This metric represents the size of a feature. LOC metric can be measured by counting the number lines of feature code that are linked to a feature expressions. LOC metric gives us indication whether a small or a large fraction of the code base is variable [28]. We assume that this metric can reduce testing effort by starting testing products that have features with a large fraction of code.

Scattering Degree (SD) and Tangling Degree (TD).

The SD metric is the number of the occurrences of a feature in different feature's expressions. This metric is measured by extracting feature names from feature expressions and calculate the average and standard deviation per product of all occurring feature. SD metric tells us about the complexity of feature implementations [28].

TD metric is the number of different features that occur in a feature expression. A lower TD is better, because high TD of features in feature expressions may impair program comprehension [28]. The goal of measuring these metrics is to measure the complexity of products and give higher priority to the products with higher complexity.

Average Nesting Depth of features (AND).

The AND metric represent the average nesting depth. AND metric can be measured by calculating the average and the standard deviation of all features and computing, based on these values, the average and standard deviation for product. Since nested features form feature expressions, this metric is useful for discussions on program comprehension [28]. This metric may give indication that the products with high average nesting depth is more complex than others. We will start testing the product with higher complexity hoping that it is most likely to contain faults.

5.3 Research Methodology

This thesis will rely on the combination of a systematic review, simulation-based controlled experiments, and industrial case studies, which will be applied at different stages of the thesis. We use systematic reviews to synthesize the research results, to summarize the existing evidence for SPL testing, and to aid in the identification of gaps in the current research. A main portion of this research proposal is on the use of search-based techniques on SPL testing.

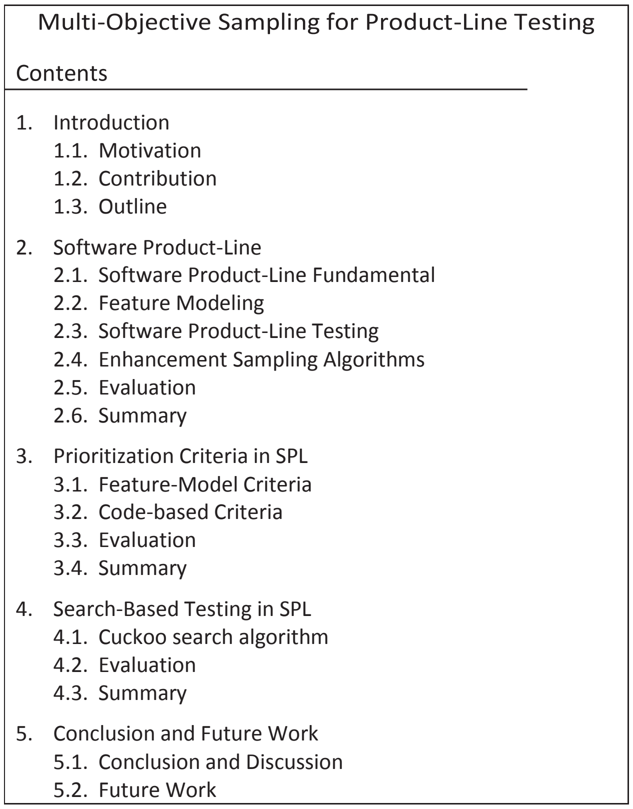
This thesis will be approached through experimental research methods. We will implement and evaluate our approaches in FeatureIDE [36]. FeatureIDE is an open-source framework based on Eclipse, which covers the whole development process of feature-oriented software and incorporates with tools for the implementation of SPLs. We will compare our work with state-of-the-art tools and algorithms that are used in SPL testing and with random generations. We will apply our approaches in a real case study to evaluate our approaches in the real world. Since our working group is the main one that design, implement, and maintain FeatureIDE, we hope to be able to perform a case study with the industrial use of FeatureIDE.

6. WORKING PLAN

In this section, we present: an outline of thesis structure, the progress status at the moment of writing this proposal, the remaining work that needs to be done to answer our research questions, and finally the publication plan of our PhD project.

6.1 Thesis Structure

As it illustrated in *Figure* 3, the thesis structure will be as follows. In Chapter 1, we plan to introduce a motivation to our work. Then, we will present a brief summary of our contributions and outline of the thesis structure. In Chapter 2, at first, we plan to present a background about SPL fundamental. Second, we will describe the properties of feature model and its constraints. Third, we will present the challenges of SPL testing and to discuss the state of the art of sampling algorithms and tools which use to sample the configurations from feature models. Finally, we are going to study the pros and cons of each algorithm and investigate how we can enhance their efficiency w.r.t. increasing their scalability to handle large feature models as well as increasing the early rate of fault detection.


Multi-Objective Sampling for Product-Line Testing

Contents

1. Introduction
   1.1. Motivation
   1.2. Contribution
   1.3. Outline
2. Software Product-Line
   2.1. Software Product-Line Fundamental
   2.2. Feature Modeling
   2.3. Software Product-Line Testing
   2.4. Enhancement Sampling Algorithms
   2.5. Evaluation
   2.6. Summary
3. Prioritization Criteria in SPL
   3.1. Feature-Model Criteria
   3.2. Code-based Criteria
   3.3. Evaluation
   3.4. Summary
4. Search-Based Testing in SPL
   4.1. Cuckoo search algorithm
   4.2. Evaluation
   4.3. Summary
5. Conclusion and Future Work
   5.1. Conclusion and Discussion
   5.2. Future Work

Figure 3: Outline of the Thesis Structure

In previous works [16, 19, 34], a limited number of feature model criteria are used to select and/or prioritize products. However, we think there is a lack of research about other criteria that may enhance SPL testing. Hence, we plan to investigate this issue further in Chapter 3. We plan to find out what metrics at code-based can help to prioritize or even select a set of products to test, since to our knowledge, no empirical research exists addressing the question of what and how code-based metrics can enhance the efficiency of SPL testing. At the end of Chapter 3, we plan to investigate other information that may help in testing such as, domain knowledge and specifications.

In Chapter 4, we plan to present search-based testing approaches to use them in selecting and prioritizing the products based on on the inputs to the fitness function. Then, we will describe the implementation and evaluation of each approach. In the last chapter, we summarize the overall approach and answers to all the listed research questions. At the end of Chapter 5, we present some open questions and outlook for further research on SPL testing.

6.2 Progress status

At the moment, we have addressed RQ1 from our research questions [1]. We have proposed similarity-based prioritization to prioritize the configurations. The input to similarity-based prioritization can be all the valid configurations if the feature models are small; if not, the input to our approach can be a set of sampled configuration created by sampling algorithms or a set of configurations created by domain experts. We selected well known sampling algorithms and integrated them to FeatureIDE. We combined our approach to these sampling algorithms. We implemented and evaluated our approach in FeatureIDE. We compared our approach to the defaults outcomes of these sampling algorithms as well as to the random orders.

The results indicate that the default order of sampling algorithms can be improved by combining our approaches to prioritize their outcomes. In addition, our approach outperforms the random orders with respect to the early rate of faults detection.

6.3 Remaining Work

As stated in Section 6.2, we have addressed RQ1 from research questions. Although our previous experiment [1] suffers from a threat to validity since we simulate the faults and we assumed that the faults are equally distributed over the features and their interactions in an SPL. Hence, we plan to evaluate our previous work by using real test cases and real SPL or generate faults into real source code of existing SPLs using techniques from mutation testing.

We are currently starting work on RQ2. After that, we are going to address RQ3. The plan is to complete the dissertation during 2016-2017.

6.4 Publication Plan

Since the remaining work is the large part of our research, we plan to submit to several conferences, and journals related to the thesis topic in the following two years. At first, we plan to extend our work from a previous work [1], to include other approach and improve the evaluation part especially the part of fault generator. We will submit the paper to a journal specialized in software testing or SPL. Second, we plan for another conference submission about feature-model and code-based metrics. We want to investigate whether this criteria can enhance the efficiency of SPL testing. Third, we plan for a conference submission to investigate using the code-based and feature-model criteria to help in selecting and prioritizing products. Finally, we are going to submit a paper to a journal where we can show the results of applying and evaluating our approach to a real case study.

7. CONCLUSION

It is a difficult task to test every product of an SPL because the combinatorial explosion of the number of products. *Combinatorial interaction testing (CIT)* is used to reduce the number of products under test and to detect the faults are caused by the interaction between a few features. The challenge in CIT is that it does not scale well for large feature models.

In this proposal, we suggest using metaheuristic search techniques to generate the products to test. Using metaheuristic search techniques, we can generate and prioritize these products. We think that using metaheuristic search techniques can increase the efficiency of SPL testing. Furthermore, we believe that there is a lack of research using code-based criteria to prioritize products, hence, we want to investigate what are code-based metrics that may increase the efficiency of SPL testing

8. ACKNOWLEDGMENTS

I thank my supervisor Gunter Saake for his support as well as Thomas Thüm and Reimar Schröter for helpful comments on a draft. This thesis proposal has been accepted for presentation at SPLC 2014 Doctoral Symposium.

9. REFERENCES

[1] M. Al-Hajjaji, T. Thüm, J. Meinicke, M. Lochau, and G. Saake. Similarity-Based Prioritization in Software Product-Line Testing. In *Proc. Int'l Software Product Line Conf. (SPLC)*. ACM, 2014. To appear.

[2] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.

[3] E. Bagheri and D. Gasevic. Assessing the Maintainability of Software Product Line Feature Models Using Structural Metrics. *Software Quality Journal (SQJ)*, 19(3):579–612, 2011.

[4] H. Baller, S. Lity, M. Lochau, and I. Schaefer. Multi-Objective Test Suite Optimization for Incremental Product Family Testing. In *Proc. Int'l Conf. Software Testing, Verification and Validation (ICST)*, pages 303–312. IEEE, 2014.

[5] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 7–20. Springer, 2005.

[6] L. Bianchi, M. Dorigo, L. Gambardella, and W. Gutjahr. A Survey on Metaheuristics for Stochastic Combinatorial Optimization. *Natural Computing*, 8(2):239–287, 2009.

[7] I. D. Carmo Machado, J. D. McGregor, and E. S. De Almeida. Strategies for Testing Products in Software Product Lines. *SIGSOFT Software Engineering Notes*, 37(6):1–8, 2012.

[8] M. Cohen, P. Gibbons, W. Mugridge, and C. Colbourn. Constructing Test Suites for Interaction Testing. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 38–48. IEEE, 2003.

[9] M. B. Cohen, M. B. Dwyer, and J. Shi. Coverage and Adequacy in Software Product Line Testing. In *Proc. Int'l Symposium in Software Testing and Analysis (ISSTA)*, pages 53–63. ACM, 2006.

[10] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction Testing of Highly-Configurable Systems in the Presence of Constraints. In *Proc. Int'l Symposium in Software Testing and Analysis (ISSTA)*, pages 129–139. ACM, 2007.

[11] M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. *IEEE Trans. Software Engineering (TSE)*, 34(5):633–650, 2008.

[12] X. Devroey, G. Perrouin, M. Cordy, P.-Y. Schobbens, A. Legay, and P. Heymans. Towards Statistical Prioritization for Software Product Lines Testing. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 10:1–10:7. ACM, 2014.

[13] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann. Understanding Linux Feature Distribution. In *Proc. of Workshop on Modularity in Systems Software (MISS)*, pages 15–20. ACM, 2012.

[14] E. Engström and P. Runeson. Software Product Line Testing - A Systematic Mapping Study. *J. Information and Software Technology (IST)*, 53:2–13, 2011.

[15] A. Ensan, E. Bagheri, M. Asadi, D. Gasevic, and Y. Biletskiy. Goal-Oriented Test Case Selection and

Prioritization for Product Line Feature Models. In *Proc. Int'l Conf. on Information Technology:New Generations (ITNG)*, pages 291–298. IEEE, 2011.

[16] F. Ensan, E. Bagheri, and D. Gasevic. Evolutionary Search-Based Test Generation for Software Product Line Feature Models. In J. Ralyté, X. Franch, S. Brinkkemper, and S. Wrycza, editors, *Proc. Int'l Conf. Advanced Information Systems Engineering (CAiSE)*, volume 7328 of *Lecture Notes in Computer Science*, pages 613–628. Springer, 2012.

[17] R. W. Hamming. Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160, 1950.

[18] H. Hemmati and L. Briand. An Industrial Investigation of Similarity Measures for Model-Based Test Case Selection. In *Proc. Int'l Symposium Software Reliability Engineering (ISSRE)*, pages 141–150. IEEE, 2010.

[19] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon. Multi-Objective Test Generation for Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 62–71. ACM, 2013.

[20] A. Hervieu, B. Baudry, and A. Gotlieb. PACOGEN: Automatic Generation of Pairwise Test Configurations from Feature Models. In *Proc. Int'l Symposium Software Reliability Engineering (ISSRE)*, pages 120–129. IEEE, 2011.

[21] M. F. Johansen, Ø. Haugen, and F. Fleurey. Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In *Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MODELS)*, pages 638–652. Springer, 2011.

[22] M. F. Johansen, Ø. Haugen, and F. Fleurey. An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 46–55. ACM, 2012.

[23] M. F. Johansen, Ø. Haugen, F. Fleurey, A. G. Eldegard, and T. Syversen. Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines. In *Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MODELS)*, pages 269–284. Springer, 2012.

[24] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990.

[25] C. H. P. Kim, D. Batory, and S. Khurshid. Reducing Combinatorics in Testing Product Lines. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 57—68. ACM, 2011.

[26] M. Kowal, S. Schulze, and I. Schaefer. Towards Efficient SPL Testing by Variant Reduction. In *Proce. Int'l workshop on Variability & composition (VariComp)*, pages 1–6. ACM, 2013.

[27] D. R. Kuhn, D. R. Wallace, and A. M. Gallo Jr. Software Fault Interactions and Implications for Software Testing. *IEEE Trans. Software Engineering (TSE)*, 30(6):418–421, 2004.

[28] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 105–114. IEEE, 2010.

[29] P. McMinn. Search-based Software Test Data Generation: A Survey: Research Articles. *Software Testing, Verification and Reliability (STVR)*, 14(2):105–156, 2004.

[30] L. Northrop and P. Clements. A Framework for Software Product Line Practice, Version 5.0. *SEI*, 2007.

[31] S. Oster, F. Markert, and P. Ritter. Automated Incremental Pairwise Testing of Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 196–210. Springer, 2010.

[32] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. Pairwise Testing for Software Product Lines: Comparison of Two Approaches. *Software Quality Journal (SQJ)*, 20(3-4):605–643, 2012.

[33] 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 *Proc. Int'l Conf. Software Testing, Verification and Validation (ICST)*, pages 459–468. IEEE, 2010.

[34] A. B. Sanchez, S. Segura, and A. Ruiz-Cortes. A Comparison of Test Case Prioritization Criteria for Software Product Lines. In *Proc. Int'l Conf. Software Testing, Verification and Validation (ICST)*, pages 41–50. IEEE, 2014.

[35] J. Shi, M. B. Cohen, and M. B. Dwyer. Integration Testing of Software Product Lines Using Compositional Symbolic Execution. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*, pages 270–284. Springer, 2012.

[36] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming (SCP)*, 79(0):70–85, 2014.

[37] T. Thüm, C. Kästner, S. Erdweg, and N. Siegmund. Abstract Features in Feature Modeling. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 191–200. IEEE, 2011.

[38] E. Uzuncaova, D. Garcia, S. Khurshid, and D. Batory. A Specification-Based Approach to Testing Software Product Lines. In *Proc. European Software Engineering Conf. and the ACM SIGSOFT (ESEC)*, pages 525–528. ACM, 2007.

[39] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. TimeAware Test Suite Prioritization. In *Proc. Int'l Symposium in Software Testing and Analysis (ISSTA)*, pages 1–12. ACM, 2006.

[40] X.-S. Yang and S. Deb. Cuckoo Search via Levy Flights. In *World Congress on Nature Biologically Inspired Computing (NaBIC)*, pages 210–214, 2009.

[41] S. Yoo and M. Harman. Pareto Efficient Multi-objective Test Case Selection. In *Proc. Int'l Symposium in Software Testing and Analysis (ISSTA)*, pages 140–150. ACM, 2007.