Improving the Testing and Testability of Software Product Lines

Isis Cabral, Myra B. Cohen, and Gregg Rothermel
{icabral,myra,grother}@cse.unl.edu

Department of Computer Science,University of Nebraska-Lincoln

Abstract. Software Product Line (SPL) engineering offers several advantages in the development of families of software products. There is still a need, however, for better understanding of testability issues and for testing techniques that can operate cost-effectively on SPLs. In this paper we consider these testability issues and highlight some differences between optional versus alternative features. We then provide a graph based testing approach called the FIG Basis Path method that selects products and features for testing based on a feature dependency graph. We conduct a case study on several non-trivial SPLs and show that for these subjects, the FIG Basis Path method is as effective as testing all products, but tests no more than 24% of the products in the SPL.

1 Introduction

Software product line (SPL) engineering has been shown to improve both the efficiency of the software development process and the quality of the software developed, by allowing engineers to systematically build families of products with well defined and managed sets of re-usable assets [4]. A large body of research on SPL engineering has focused on reuse of core program assets [4, 15, 17], refined feature modeling [8, 9, 23], and code generation techniques [2, 7]. There has also been research on testing software product lines [3, 6, 10, 24].

Despite this prior work, there still remains a need to improve reuse during the software testing process. Kolb and Muthig [15] point out that testing has not made the same advances as other parts of the SPL lifecyle and remains a bottleneck in SPL development. Their work highlights issues related to testability of SPLs, where testability is viewed as the ease with which one can incorporate testing into the development cycle and increase reuse while retaining a high rate of fault detection. They comment that the primary strength of SPL development, variability, also has the greatest impact on reducing testability [15], due to the combinatorial explosion of feature combinations that occurs as variability increases [6, 17]. In other related work, Jaring et al. [13] point out that the testability of a product line can be viewed as a function of the binding time of variability, and that providing early binding can increase the ability to test products early. McGregor [17] and Cohen et al. [6] have suggested ways to reduce the combinatorial space by sampling products for testing using combinatorial interaction testing [5]; this work does not address testability issues.

While all of this work aims at the core problem of software product line testing, none of it specifically considers reuse by examining the feature model and analyzing testability at a finer grain. In this work we consider testing from this perspective. We drill down into the issue of variability and analyze different types of variability, e.g. optional features versus alternative choice features, as they relate to testability. We conjecture that while the alternative choice features have a negative impact on testability by increasing the number of products, optional choice features do not. We then propose a new black box approach for testing software product lines that attends to these issues. We hypothesize that our approach can reduce testing effort while retaining good fault detection in the presence of alternative features.

Our approach, which we call the FIG Basis Path method, translates a feature model into a feature inclusion graph that is, in essence, a feature model dependence structure. We associate all features with sets of test cases and then walk this graph to generate a subset of independent paths (or products) that cover the graph for testing. This is analogous to the basis path approach for testing software applications [26] which finds a "possibly minimal" set of paths to cover all nodes in a program's control flow graph.

We report results of a case study on two software product lines. In both SPLs we can achieve the same fault detection results as we can testing all products. Further analysis shows that we can also use a grouped variant of the Basis Path algorithm to test subfamilies of the SPL as defined by the alternative features.

2 Background and Related Work

There has been a lot of work on feature modeling of which we present only a small subset [1,2,14,19,21]. In a feature model, a product line can be represented by mandatory and variable features. The variable features may be optional or alternative choices. In their simplest form alternative choice features allow for an exclusive or relationship. We can also have cardinalities assigned that allow for $0...n$ or $1...n$ relationships where the first number is the lower bound and the second number is an upper bound. An exclusive or alternative feature is usually the default for alternative features in a model and is a 1...1 relationship.

We present a small product line to help to explain these ideas and to illustrate our techniques. The feature model for this product line (shown using the Orthogonal Variability Model (OVM)) [19] is seen in Figure 1 on the left (the right portion of this figure will be discussed later). This product line defines a family of 42 calculator programs. It has a core set of features, Exit and Clear and one optional feature, Backspace. Users can select one of three languages (English, Chinese or Spanish) used in the menus, titles and help. Finally the memory features include Memory Store and Recall.

The mandatory features in our sample product line are Core and Language. Within each of these there is some variation. Store is an Or feature from Memory and it is required only if the Memory Recall feature is selected.

Feature models have been used for generative programming [2, 7, 23], providing a model based approach to the realization of product lines. Feature models

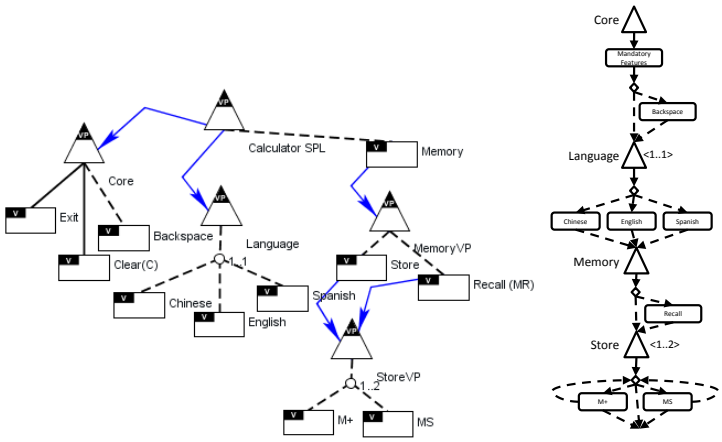

Fig. 1. Calculator SPL Feature Model

have also been used to model the product space for instantiating products for testing [3,6,24]; for instance, the work of Uzuncaova et al. [24] transforms the feature model into an Alloy specification and uses this to generate test cases, while the work of Cohen et al. [6] uses the feature model to define samples of products that should be included in testing. Similarly, the PLUTO methodology [3] uses the feature model to develop extended use cases that contain variability which can formulate a full set of test cases for the family of products. Schürr et al. [22] use a classification tree method for testing from the feature model. Other extensions of feature models have been for staged generation [8] or modeling constraints [9]. None of this work explicitly uses the feature model as we do, in a graph based representation, that can be used to select products (and test cases) for testing. The work of Bachmeyer et al. [1] also uses a graph based representation of a feature model, but they do not use this in the testing process.

Other work on software testing product lines includes that of Denger et al. [10] who present an empirical study to evaluate the difficulty of detecting faults in the common versus variable portions of an SPL code base concluding that the types of faults found in these two portions of the code differ. They use both white and black box techniques but do not test from the feature model.

3 Leveraging Redundancy for Testing via Feature Models

We begin with the conjecture that black box testing of software product lines can be made more efficient and effective by designing the product line architecture (and resulting feature model) in a manner that supports reuse of product line testing results across different products. Others have argued that variability decreases testability [15], but we believe that there should be a finer grained examination of this argument. Both optional and alternative features can be viewed as points of variability in a software product line, yet we believe they

may behave differently from a black box testing approach and provide different opportunities to reduce testing effort, as we explain next.

Our methodology involves the following steps: (1) transform the feature model into a *feature inclusion graph*; (2) associate use cases with each feature; (3) develop test cases for each use case; (4) Select basis paths on this graph; and (5) for each path (product), run all test cases for the included features.

3.1 Feature Inclusion Graph

In this section we present a transformation of the feature model into a graph that we call a *feature inclusion graph* (FIG), which represents feature dependencies derived from the feature model. In a FIG, all features that appear on a non-branching path are included in the same product, while branches represent the variability in feature composition. We view the FIG as having a loose connection to the control flow graphs used in software testing; a control flow graph shows explicit flow of control in a program and can be used to select test cases for white box testing. Harrold [12] has suggested that regression testing techniques can be applied to different abstractions of software artifacts as long as they can be represented as a graph and tests can be associated with edges. In our scenario we do not have control flow; rather, our paths represent a combination of features and its dependencies, but we use a common method from control flow based testing to find a *basis path set* [26] for the graph – a set of independent paths through the program graph.

The FIG contains all features of the SPL. We next show how it is derived using different parts of a feature model from OVM [19]. In OVM, the core concepts of an SPL model are the variation points and variants. Each variation point (VP) has at least one variant and the edges between VPs and variants indicate dependencies. In a FIG we apply the same OVM concepts. A FIG has two main components, features and edges. The edges represent the variability of our diagram making explicit all possible paths that we can traverse to generate the minimum set of products. The features are classified as Mandatory, Optional, or Alternative. Next, we describe how each feature and edge are represented in the FIG and how they interact with each other.

In a FIG, a variation point is represented as a triangle and variants are represented as rounded rectangles. A Mandatory dependency is represented by a solid edge between a VP and variant, while an optional dependency uses a dashed edge. A diamond represents the variability of optional and alternative features. Figure 2 shows an example of Mandatory and Optional features represented in the OVM language and FIG diagram, respectively. In this example we see on the left (Figure 2) two mandatory features in OVM (B and C). These are both required in the same *flow of control* therefore we put them on a single non-branching edge of our FIG (lower bottom of figure). Note that either B or C can come first since the dependency is only important at the branches (e.g. this is a partial order). On the right (Figure 2) we show two optional features (again B and C). Here we have added three branched edges. The middle edge represents the case in which neither feature is included, while each of the other

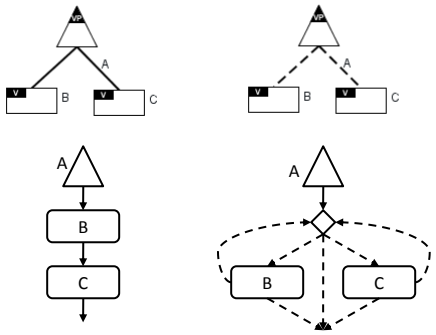

Fig. 2. Mandatory and Optional Features

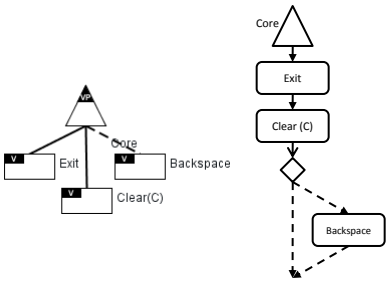

Fig. 3. Calculator Example

edges allow for the inclusion of either feature. We also include a back edge for each feature since it is possible to include a second feature. Assuming that we allow only one instance of a feature for a single product, we can see that there are four possibilities in this graph: we can have no optional features, one of B or C, or both B and C.

From a testability perspective we view this type of variation to be more testable than some other types of variation, since we can include both features (B and C) in a single product. With two optional features we have a 75% reduction in the number of products that we must instantiate in order to test all features. We can apply this to the Core Variation Point and its variants in the calculator SPL. The Core Variation Point has two mandatory features (Exit and Clear (C)) and one optional feature shown in Figure 3. As we can see in this case the optional feature (backspace) can be included in the first product tested providing us with a single instance.

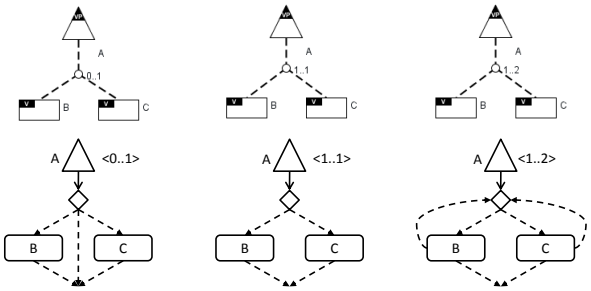

Fig. 4. Alternative Features

Alternative features are features that are mutually exclusive and present a more difficult challenge for testability. We argue that these are the true deterrents to testability since only one feature can be present in an SPL at a time. Even with these types of variation points we may still gain some benefit in reusability. Figure 4 shows three examples of alternative features in OVM (top)

and its corresponding FIG (bottom). The first example has cardinality 0...1, i.e., this is really an optional feature and we can include at most one of the two alternatives. In this case we expect to see a small benefit from the optional feature characteristics. We need two of the three possible products to cover all features.

The second example (middle) shows the exclusive or 1...1 relationship. This is the least testable type of variation since it forces the combinatorial space to increase. Here we have two dashed edges to B and C, no back edges and no middle edge. We have two possible products and need to test both to cover the features of this graph. We see no reduction.

The last example (right) is when we have a 1...n relationship; the figure shows a 1...2 relationship. We have a back edge from each feature, and we can cover all features using a single product even though there are three products (B, BC, and C), by including both B and C in our product for testing.

The graphs do not explicitly incorporate constraints in the representation. We maintain a separate set of constraints that we can check during our graph traversal, to ensure consistency, but will examine this in future work.

3.2 Selection Algorithms

In this section we present four methods for selecting products for testing. The first two use the FIG and the second two do not. The first algorithm is our core algorithm called the *FIG Basis Path* algorithm. The idea is to select a set of independent paths in the program that cover all features in the graph. We then present a variant of this called the *FIG Grouped Basis Path* algorithm, that tests subfamilies of the product line grouped by the alternative features in the SPL. We believe that this algorithm will be incorporated into the development process more smoothly, where one particular subfamily is created at a time. The third algorithm does not use the FIG, but is used in our empirical comparison as a method that we believe will be less expensive; we call this the *All Features* algorithm. This algorithm greedily chooses products until all features in the product line have been included at least once. The last method we discuss is also used as a basis for comparison. We expect that it will be stronger than the other comparison method, but also perhaps more expensive. This is the *Covering Array* method suggested by McGregor [17] and Cohen et al. [6]. In this method we select a subset of products from the feature model that cover all pairs of features in the SPL. We describe each method in more detail next.

The **FIG Basis Path Algorithm** (Algorithm 1) is based on the basis path algorithms in [26]. In this algorithm we assume that the FIG is built and that we have a set of constraints on the features. We begin by setting the basis path set (BP) to be empty. We then iterate through all paths in the FIG in a depth first traversal order (to ensure we find the longest paths first).In the algorithm we reference the authors use a breadth first search, but our objective is slightly different. For each path we check the constraint set to see if the path is feasible. If it is, we then check if it is linearly independent with the other paths in BP. (In our study we performed this step manually, however, it can be automated with a constraint solver.) If it is independent we add it to BP. For example,

Algorithm 1 FIG Basis Path Algorithm

```
BasisPath(FIG)
BP = ∅
for all paths, P, ∈ FIG (using DFS order) do
   if P is feasible then
      if LinearlyIndependent(P, BP) then
         add P to BP
      end if
   end if
end for
```

suppose we want to select the minimum set of products in the calculator SPL. We show the FIG on the right portion of Figure 1. For each path, we evaluate if it is feasible by checking existing constraints. In this case, all paths that include the Memory Recall variant and do not include the Store variant will be removed from the final set of paths (Products). We next begin our selection. In the first path, 6 variants are selected, containing all of the mandatory features (Exit, and Clear (C)) and optional features (Backspace), one language - Spanish - and two variants from the Memory variant point: Store (M+) and Recall. The second path substitutes the Store variant from the previous path (M+) to MS. The third and fourth paths change only the Language variant.

The **FIG Grouped Basis Path Algorithm** is a modification of the FIG Basis Path algorithm, in which subfamilies of the product line are grouped based on the alternative features. We begin by generating all of the paths in the FIG in depth first order and check each for feasibility. We then group all feasible paths by alternative feature groups, where all paths that include a particular alternative feature are included in its group. If there are paths that contain no alternative features, we create an additional group to hold these. For example, suppose we want to group based on the language VP in the calculator SPL. In this case we would find all paths that contain Spanish and put them into one group. All that contain Chinese go into in another, and the rest that contain English are put into another. Once we have the grouping, we use the Basis Path algorithm for each group where the FIG is replaced with the set of paths in the group. We can skip checking feasibility since this has already been performed.

Our third algorithm, the **All Features Algorithm**, does not require a FIG. This algorithm is less expensive than the first two because it does not involve enumerating paths and walking the graph. Instead its goal is to include a set of products that just cover all features. We begin by placing all features into one of two sets, Mandatory and all others. We include additional constraints to enforce our alternative features, then we order features in descending order based on the the number of constraints on that feature. We keep a set we call *used features* which starts as empty. For each product, we greedily add features (putting them in the constraint order described) into a product, skipping those that are already in the used feature set (unless mandatory or part of a requires constraint), or that violate a constraint, until we have a product including the greatest number of unused features. We then update the used features set. Once all features have been included in at least one product we are done. For the calculator SPL,

we create 3 products. The first product contains the mandatory features, the Chinese language and all variants associated with the Memory variant. The second and third products do not include any variant from Memory, but the Language variant is changed to English and Spanish, respectively.

Our fourth algorithm, the **Covering Array Method** uses a pair-wise approach and covers all pairs of features in at least one product. This technique tests interactions between features and has been shown to be effective in testing many types of configurable software [20]. The base object used to select the sample is a covering array. A few differences can be noted between this and our other methods. First, in the Covering Array method we consider optional features as being both included and not included. Therefore we would not be able to simply test a product with both A and B but would need to test A with and without B, and B with and without A as well as neither feature. While possibly a stronger testing criterion we expect that this method will be more expensive and may not be helped by improved testability as we have described it.

4 Case Study

To gain insights into the operation of the FIG basis path approach we conducted a case study, comparing the approach to the three other approaches described in Section 3. Our goal is to address the following research questions:

RQ1: How does the FIG basis method compare with other test methods?
RQ2: Can we reduce the effort required to test groups of products through the grouped basis method?

4.1 Study Objects

As objects of study we selected two software product lines, both developed by other researchers and used in previous studies of SPLs. The first SPL is a Graph Product Line (GPL) created by Lopez-Herrejon and Batory [16]; it is built using the AHEAD methodology and implemented as a series of .jak files [2] (an extension of the Java language). The second SPL is a Mobile Software Product Line [11] created by Lancaster University and widely used in previous studies.

Table 1 lists, for each of our software product lines, the total number of lines of code excluding comments (*LOCs*), the number of classes present (*Classes*), the number of products that can be created (*Products*), the number of faults present (*Faults*), the number of variants classified as Optional, Alternative, and Or (*Variants*) and the number of constraints classified as Require and Exclude (*Constraints*). The total number of lines of code (LOCs) corresponds to the product that has the most features selected.

The Graph Software Product Line (GPL) is an SPL that implements a family of graph algorithms in which each product is a type of graph. The code base includes 1435 lines of jak code and consists of 15 features. A graph is either directed or undirected. Edges can be weighted or unweighted. A graph product requires at most one search algorithm: depth-first search (DFS) or breath-first

Table 1. Objects of Study

					Variants			Constraints	
	LOCs	Classes	Products	Faults	Opt.	Alt.	Or	Require	Exclude
GPL	1435 (jak)	12	38	60	0	4	1	10	1
MobileMedia - V5	2220	37	16	10	4	0	0	0	0
MobileMedia - V6	2173	38	24	10	4	0	1	4	0

search (BFS), and at most one or more of the following algorithms: vertex numbering, connected components, strongly connected components, cycle checking, minimum spanning tree and single-source shortest path. The GPL feature model contains a total of 80 instances without constraints. After reading the documentation for the GPL we created a feature model for it, as shown in Figure 5. To create this model we needed to resolve some ambiguity in the documentation and we also reduced the possible cardinality for combinations for the variant point Alg. Ultimately we obtained 38 possible instances of the product.

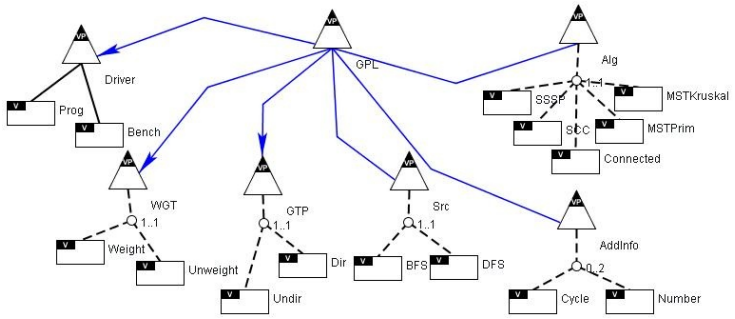

Fig. 5. Graph SPL Feature Model

Mobile Media is an SPL that implements mobile applications that manipulate media (photos, music and video) on mobile devices. Mobile Media has evolved since 2005 to support several types of media. Mobile Media has nine releases implemented in two paradigms: aspect oriented and object oriented.

For our study, we selected two versions of Mobile Media that were developed using the object oriented paradigm, versions 5 and 6. In version 5, users are allowed to manipulate image files in different mobile devices as well as send messages, set favorite pictures, copy images and perform other operations. Version 6 is a refactored version of version 5 and it includes one more variation point. In this version, users are allowed to manipulate two different types of media: photo and music. Both versions share a few operations but they have different underlying code bases due to the refactoring. The Mobile Media Feature Model allows us to derive a total of 16 and 24 instances for version 5 and 6, respectively. Figure 6 presents the feature model for Mobile Media and its evolution.

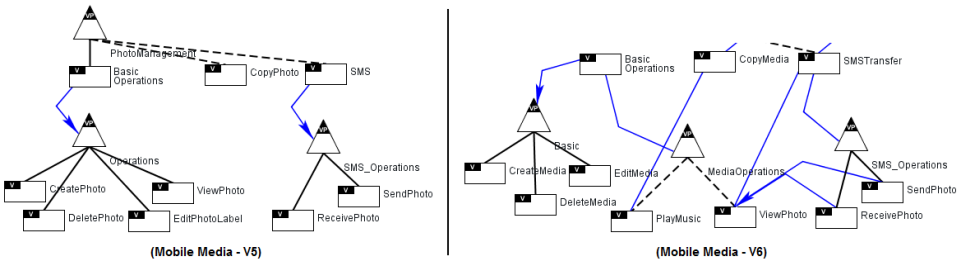

Fig. 6. Mobile Media SPL Feature Model

4.2 Test Suites

To conduct our study test suites were developed by associating each feature with its correlated use case requirements. For each feature, we developed concrete test cases that cover the primary scenario as well the alternatives use cases. We used the documentation provided with the object to generate these. All test cases were created by other researchers in our group not including the authors of this paper. For the GPL product line, the test suites are command line test cases. For Mobile Media the test cases are GUI based and implemented using a combination of two open source testing tools, Microemulator [18] and Abbot [25].

4.3 Fault Seeding

In the Mobile Media application, during the course of working with the system, we found 10 actual faults that caused the system to working improperly.We corrected each fault based on the requirements provided with the application and then re-seeded each fault into a single *faulty version*. We thus had 10 faulty versions of this application for both version 5 and 6.

For the GPL application, we provided six students in our laboratory, who were not involved with the study itself and had no knowledge of the approaches being studied, with a document on an approach for doing fault seeding and subsets of the .jak files. We asked each student to seed 10 faults into their set of files. This yielded 60 faulty versions of this application.

4.4 Study Conduct

To conduct our study we applied each of the four testing approaches to each of our fault free objects. We executed these test cases on our faulty versions, and to determine whether a test case detected a fault, we compared the output of the faulty version under that test with the output of the original (non-faulty) version of the object under that test. All of our executions were performed on a 1.8GHz Intel Pentium M with 1GB of system memory running SuSE Linux 10.1 platform equipped with the Java 1.5 JDK.

5 Results

In this section we examine the results of our research questions. We begin with RQ1 which asks how the FIG Basis Path method compares with other methods. Table 2 shows the data for both applications. The first column shows the number of products tested, followed by the number of test cases run. The rightmost column shows the number of faults detected by each technique.

In considering this research question we examine three methods: the Covering Array method, the All Features method, and the Basis Path method, and we compare these to an All Products method which performs an exhaustive enumeration of all products. (The Grouped Basis Path method is considered for RQ2.) As the table shows, in the GPL of the 60 faults inserted, 54 were found when we tested all products. Both the Covering Array method and the Basis Path method also found 54 faults, however the Basis Path method used fewer than half as many products as the Covering Array method and 39.7% of the test cases. The least expensive method was All Features (5 products and 26 test cases); however, this technique missed 3 faults when compared with the other techniques.

Table 2. Number of Test Cases and Faults Detected by Technique: Mobile Media SPL

	Graph SPL Total Number of Products: 38 Total Number of Faults: 60		
Method	**# Products**	**Test Cases**	**Faults Detected**
Covering Array	20	141	54
All Features	5	26	51
Basis Path	9	56	54
All Products	38	256	54

	Mobile Media 5 Total Number of Products: 16 Total Number of Faults: 10			Mobile Media 6 Total Number of Products: 24 Total Number of Faults: 10		
Method	**# Products**	**# Test Cases**	**#Faults Detected**	**# Products**	**#Test Cases**	**# Faults Detected**
Covering Array	5	190	7	6	201	10
All Features	1	49	7	2	71	10
Basis Path	1	49	7	2	85	10
All Products	16	348	7	24	839	10

We next consider the results for the two versions of Mobile Media. In this case we see that all methods found all of the faults in both versions. For version 5, the All Features and Basis Path methods required only one product and 49 test cases, compared with 348 test cases for the All Products method and 190 test cases for the Covering Array method. For version 6, the All Features and Basis Path methods required only two products with 71 and 85 test cases respectively, compared with 839 test cases for the All Products method and 201 test cases for the Covering Array method. We discuss the implications of these results in the next section.

To answer RQ2 we examine data shown in Table 3. This table shows the data grouped by the alternative features of each SPL. The left side of the table

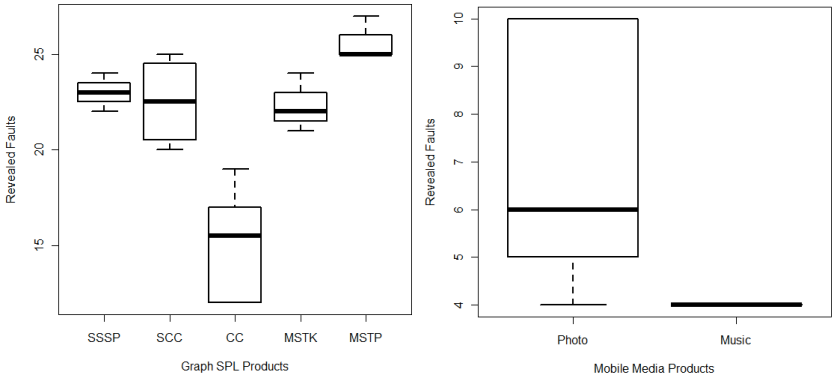

Fig. 7. Number of Test Cases and Faults Detected Grouped by Alternative Features

shows data for all feasible paths in each group, including the number of products, number of test cases, and number of faults detected. The right side of the table shows the same data for the selected products using the Basis Path method. In every group of products we see that we can reduce the number of products tested while retaining the fault detection capability. For GPL, our reduction in products ranges from 67% (CC) to 33% (Shortest, MTSP, MSTK). In addition we have used between 37% and 79% of the test cases required for all feasible paths, resulting in at least a 20% reduction in the required test cases. For Mobile Media, we had a reduction of between 71.2% and 77.7% of the test cases and 75% (Music) to 80% (Photo) of the products.

Table 3. Number of Test Cases and Faults Detected by Alternative Variants

Graph SPL						
	All Feasible Paths			Grouped Basis Path		
Variant	#Product	#TC	#Faults	#Product	#TC	#Faults
Shortest	3	19	27	2	13	27
SCC	4	34	28	2	17	28
CC	6	46	21	2	17	21
MSTP	3	14	26	2	11	26
MSTK	3	29	29	2	24	29
Mobile Media v6						
	All Feasible Paths			Grouped Basis Path		
Variant	#Product	#TC	#Faults	#Product	#TC	#Faults
Music	4	139	10	1	40	10
Photo	5	220	10	1	49	10

In an additional analysis we wanted to determine whether any subset of n paths could have been selected with the same fault detection results within each group. For GPL, we performed a pair wise comparison between products since we have selected two products for each group. For each group we combined all combinations of 2-paths and calculated the fault detection. We show this data in the form of a box plot (Figure 7). In each plot we see a range of fault detection,

indicating that the Basis Path method is providing the best fault detection (we know that it is at the top of the box plot since all basis path results provided the same fault detection as the full set of feasible paths in that group). Since the Grouped Basis Path for Mobile Media selected only one product from the whole set of feasible products, we evaluated the fault detection for all products that belongs to the same group. The data in Figure 7 shows that there is a range of fault detection between products in the Photo variant, but products with the Music variant selected have the same fault detection. This confirms that we cannot randomly select a subset of products within groups and necessarily be assured of the same fault detection.

5.1 Discussion

For RQ1, based on this data we believe that the FIG Basis Path method is efficient at finding faults and is at least as effective as other techniques. In the product line that we define as less testable due to the alternative features (GPL) we see that the Basis Path method performed the best. It found as many faults as the other techniques for 60% fewer test cases than the CA technique, and 55% fewer products. In the Mobile Media application, where we believe we have a more testable product due to the small number of alternative features, we see that the FIG Basis Path was as effective at finding faults as all other methods, and costs the same as the least expensive method, All Features. It was less expensive than the covering array method as well. Given these results we suggest that although the cost of computing the FIG Basis Path may be slightly higher than that for All Features, the technique appears to work well for both types of feature model elements (alternative and optional), therefore it is the more robust technique. Further evaluation is needed to understand the efficiency of using a FIG with higher granularity variants. We believe there is a correlation between the granularity in the feature models and the efficiency of our technique. We also have analyzed where the faults lie within our applications and many faults were located in the mandatory and optional features. We need to further analyze the impact of faults that are embedded inside of the variant portions of the code to fully understand the effectiveness of these techniques.

For RQ2, we see that it is possible to test parts of the product space more efficiently using the Grouped Basis Path method when the feature model has alternative variant points. In the GPL application, where we believe we have a less testable product due to the variability and a large number of requirements constraints, we were able to select a small set of products that revealed all our faults with fewer test cases and products. Furthermore, the boxplots tell us that we cannot simply select the paths to test randomly. This suggests a further use of the FIG Basis Path method, where we want to focus on parts of the SPL at a time or where development is taking place in stages, based on specific variation points. Conversely, the Grouped Basis Path method did not show any improvement over the FIG Basis Path method in the Mobile Media application. We believe the small number of constraints of the Mobile Media application has some influence over the method. We conclude for RQ2 that we can use FIG Grouped Basis Path to reduce test effort.

6 Conclusions and Future Work

In this paper we have used the feature model to drive test case selection and have asked if we can reduce test effort while retaining fault detection capability through a graph-based selection algorithm. Using the FIG Basis Path method we were able to detect the same number of faults as we did when testing all products, by testing as few as 6% and no more than 24% of products in our SPLs, and running only 10% of the test cases as All Products in the best case. The most effective non-graph technique, the Covering Array method, required us to test between 13% and 54% of products respectively in the same systems. In the subject with only optional features, we see that our method does as well as all other techniques in fault detection and costs no more than the least expensive technique, All Features.

In future work we plan to examine this technique on larger software product lines with more complex faults. We will also examine other variations of the feature model such as $1...n$ relationships and the impact of constraints on our model.

Acknowledgments

We thank B. Gavin, T. Yu, S. Huang, A. Sung, S. Kuttal and W. Xu for help in seeding faults and W. Motycka for developing the test suite for the subjects used in this work. We also thank Eduardo Figueiredo for providing us the source of Mobile Media Software Product Line. This work was supported in part by NSF under grants CCF-0747009 and CNS-0454203, and by the AFOSR through award FA9550-09-1-0129.

References

1. R. C. Bachmeyer and H. S. Delugach. A conceptual graph approach to feature modeling. In *Intl. Conference on Conceptual Structures*, pages 179–191, 2007.
2. D. Batory. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.
3. A. Bertolino, A. Fantechi, S. Gnesi, and G. Lami. Product line use cases: Scenario-based specification and testing of requirements. In *Lecture Notes in Computer Science*, pages 425–445, 2006.
4. P. Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley, 2001.
5. D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.
6. M. B. Cohen, M. B. Dwyer, and J.Shi. Coverage and adequacy in software product line testing. In *Workshop on the Role of Architecture for Testing and Analysis*, pages 53–63, July 2006.
7. K. Czarnecki. Overview of generative software development. In *Unconventional Programming Paradigms*, pages 313–328, 2004.

8. K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, pages 143–169, 2005.
9. K. Czarnecki, S. She, and A. Wasowski. Sample spaces and feature models: There and back again. In *Intl. Software Product Line Conference*, pages 22–31, 2008.
10. C. Denger and R. Kolb. Testing and inspecting reusable product line components: First empirical results. In *Intl. Symposium on Empirical Software Engineering*, pages 184–193, 2006.
11. E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor Filho, and F. Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *Intl. Conference on Software Engineering*, pages 261–270, 2008.
12. M. J. Harrold. Architecture-based regression testing of evolving systems. In *Workshop on the Role of Architecture for Testing and Analysis*, pages 73–77, July 1998.
13. M. Jaring and J. Krikhaar, R. L.and Bosch. Modeling variability and testability interaction in software product line engineering. In *Intl. Conference on Composition-Based Software Systems*, pages 120–129, 2008.
14. 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, Carnegie-Mellon University Software Engineering Institute, November 1990.
15. R. Kolb and D. Muthig. Making testing product lines more efficient by improving the testability of product line architectures. In *Workshop on Role of Software Architecture for Testing and Analysis*, pages 22–27. ACM, 2006.
16. R. E. Lopez-Herrejon and D. S. Batory. A standard problem for evaluating product-line methodologies. In *Intl. Conference on Generative and Component-Based Software Engineering*, pages 10–24, 2001.
17. J. D. McGregor. Testing a software product line (cmu/sei-2001-tr-022). Technical report, Carnegie Mellon Software Engineering Institute, 2001.
18. MicroEmulator. http://www.microemu.org/, 2010.
19. K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering*. Springer, Berlin, 2005.
20. X. Qu, M.B.Cohen, and G.Rothermel. Configuration-aware regression testing: An empirical study of sampling and prioritization. In *International Symposium on Software Testing and Analysis*, pages 75–85, July 2008.
21. P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: A survey and a formal semantics. In *Intl. Requirements Engineering Conference*, pages 136–145, 2006.
22. A. Schürr, S. Oster, and F. Markert. Model-driven software product line testing: An integrated approach. In *Theory and Practice of Computer Science*, pages 112–131, 2010.
23. S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Intl. Conference on Generative Programming and Component Engineering*, pages 95–104, 2007.
24. E. Uzuncaova, D. Garcia, S. Khurshid, and D. Batory. Testing software product lines using incremental test generation. In *Intl. Symposium on Software Reliability Engineering*, pages 249–258, 2008.
25. T. Wall. Abbot Java GUI test framework. http://abbot.sourceforge.net/doc/overview.shtml, 2010.
26. J. Yan and J. Zhang. An efficient method to generate feasible paths for basis path testing. *Information Processing Letters*, 107(3-4):87 – 92, 2008.