7-2010

Designing Software Product Lines for Testability

Isis Cabral
University of Nebraska – Lincoln, icabral@cse.unl.edu

DESIGNING SOFTWARE PRODUCT LINES FOR TESTABILITY

by

Isis Cabral

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professors Gregg Rothermel and Myra B. Cohen

Lincoln, Nebraska

July, 2010

DESIGNING SOFTWARE PRODUCT LINES FOR TESTABILITY

Isis Cabral, M. S.

University of Nebraska, 2010

Advisors: Gregg Rothermel and Myra B. Cohen

Software product line (SPL) engineering offers several advantages in the development of families of software products such as reduced costs, high quality and a short time to market. A software product line is a set of software intensive systems, each of which shares a common core set of functionalities, but also differs from the other products through customization tailored to fit the needs of individual groups of customers. The differences between products within the family are well-understood and organized into a feature model that represents the variability of the SPL. Products can then be built by generating and composing features described in the feature model.

Testing of software product lines has become a bottleneck in the SPL development lifecycle, since many of the techniques used in their testing have been borrowed from traditional software testing and do not directly take advantage of the similarities between products. This limits the overall gains that can be achieved in SPL engineering. Recent work proposed by both industry and the research community for improving SPL testing has begun to consider this problem, but there is still a need for better testing techniques that are tailored to SPL development.

In this thesis, I make two primary contributions to software product line testing. First I propose a new definition for testability of SPLs that is based on the ability to re-use test cases between products without a loss of fault detection effectiveness. I build on this idea to identify elements of the feature model that contribute positively and/or negatively towards SPL testability. Second, I 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. This method should increase our ability to re-use results of test cases across successive products in the family and reduce testing effort. I report the results of a case study involving several non-trivial SPLs and show that for these objects, the FIG Basis Path method is as effective as testing all products, but requires us to test no more than 24% of the products in the SPL.

ACKNOWLEDGMENTS

I would like to thank all those whose helped made this thesis' existence possible. Thank you for the encouragement, support, and for all prompt feedback during these two years.

I would like to thank my advisor Gregg Rothermel for giving me this opportunity, sharing his knowledge, and for carefully revising and commenting on all off my writing. I would like to thank my co-advisor, Myra Cohen, for the support in developing the idea that is the backbone of this thesis, from the brainstorming to its conclusion, and for making me believe in myself. I also would like to thank Dr. Lisong Xu for serving on my committee.

I would like to thank B. Gavin, T. Yu, S. Huang, A. Sung, S. Kuttal and W. Xu for the help in seeding faults, and W. Motycka for developing the test suite for the subjects used in this work. I also thank Eduardo Figueiredo for providing the source of the Mobile Media software product line.

I would like to thank all my friends in the Esquared lab. My special thanks goes to Wayne, who was always willing to discuss technical issues, and whose "50 cents" words were always worth a lot; and to Zhihong for all the laughs and the "You will be fine!".

I also would like to thank to my best friend, Debora, for being such a good friend. I am grateful to all my friends who have provided encouragement and support along the way: Katie, Tingting, Ahyoung, Matt Kohrell, Savio, Berggren family, Flavia and Flavio. Thank you all!

Lastly, I would like to thank my family for the support and encouragement at all times, as well as my husband Eduardo who has been a constant source of support throughout this journey.

I dedicate this thesis to the memory of my father, my biggest fan and my best friend.

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.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The ultimate goal of the software industry is to provide users with high quality custom applications that attend to their needs. Over the years, individual companies have developed a series of similar applications that share a lot of functionality, each customized for specific demands of particular customers. Given this trend, software industries face a new development challenge in creating and maintaining a series of related products and leveraging commonality and reuse among these products in order to keep the cost and development time down, without reducing individual application quality.

Software product line engineering (SPLE) has been shown to be a very successful approach to this type of software development allowing engineers to design and build families of products [12, 29, 56, 70]. This paradigm has received considerable attention from both industry (e.g. Siemens, Boeing, Hewlett-Packard, Philips, Nokia, and Bosch) and the software research community as it demonstrates how the development of products can be improved through managed reuse and more importantly how to respond quickly and effectively to market opportunities [12].

Clements and Northrop [12] defined a software product line (SPL) as "a set of

software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular segment or mission and that are developed from a common set of core assets in a prescribed way." Core assets represent a variety of reusable artifacts used in all parts of the software lifecycle and incorporate all the variability of an SPL. These assets are composed of requirement models, architectural and design models and test models as well as the code base itself. Feature diagrams, widely used in SPLE, represent graphically the variability of SPLs facilitating the management and design decisions performed by the engineer team. Software product line engineering differs from development of single systems in two aspects. First, the development is divided into two main processes: domain engineering which defines the entire family of products, and application engineering where individual products are instantiated and built. Second, the variability is explicitly modeled and managed. We will discuss these differences in Chapter 2.

A large body of research on SPL engineering has focused on reuse of core program assets [12, 38, 44], refined feature modeling [17, 20, 67], and code generation techniques [3, 18]. There has also been research on testing software product lines [7, 15, 21, 50, 58, 61, 69]. Software testing is a challenge for SPLE. Testing for single systems can consume up to 50% of the total cost of system development. For an SPL, this number is even higher due to the shortened development time of software systems. Software quality is an essential factor for the success of software companies. Achieving high quality in an SPL is crucial since a single failure can be propagated into multiple products at the same time and it is practically impossible to test all products completely.

Kolb and Muthig [38] point out that testing has not made the same advances as other parts of the SPL lifecycle 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 [38], due to the combinatorial explosion of feature combinations that occurs as variability increases [15, 44].

While this research points 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. This thesis considers testing of SPLs from this perspective. We believe that testing of SPLs can be made more efficient and effective by designing the feature model of a family of products in a way that supports the testing of different products. We next present some motivation to show some challenges faced in testing SPLs.

1.1 Example and Motivation

In this section we present an example of a software product line that will be used throughout the rest of this thesis to illustrate our ideas. We chose to model a calculator due its simplicity and understandability.

The Calculator SPL is a software product line that defines a family of 144 different calculator programs, each varying in the combinations of features that it includes. The calculator program has a common set of basic operations that will be included in all *variants* (instances of the SPL). These include addition, subtraction, division and multiplication operations, as well as a core set of features Exit, Clear (C) and Clear Entry (CE). It also provides a set of advanced operations such as Percentage, Square Root, Reciprocal and Sign. These are each optional features. The calculator supports three different languages (English, Chinese or Spanish) to be used in the

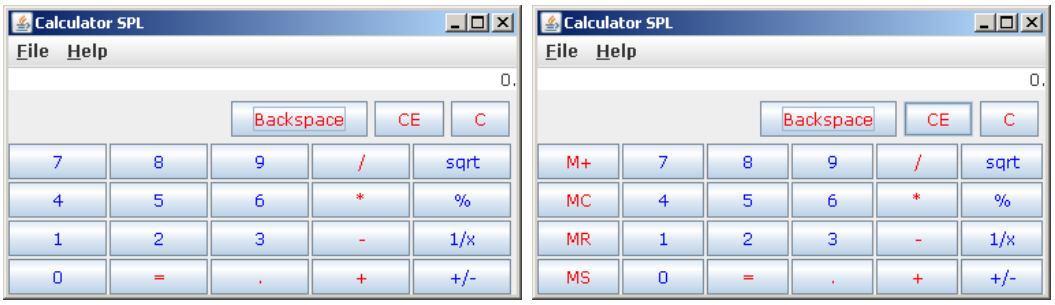

Figure 1.1: Calculator programs

menus, title and for help. An individual calculator supports a single language. Other optional features are the backspace operation and the memory capabilities. Users can store numbers into memory in two different ways as well as clear and recall stored numbers from memory. The feature model for the Calculator SPL is introduced in Section 2.2.

Figure 1.1 shows us examples of two calculator programs that belong to the same software product line. The program on the left (Program 1) supports the most basic operations only and uses English as its language. The program on the right (Program 2) also uses English but contains all of the memory features. By simply changing the language we can generate 4 more caculator programs that vary slightly from the ones shown. We can also create a range of products with differing combinations of the optional features.

The Calculator SPL has 19 features; a small number when compared with real software product lines but complex enough for testing. To understand the complexity of testing SPLs, assume that we need to test both programs based on their specifications. Table 1.1 lists the total number of tests defined for each core function as well as the features selected in each program. If we look carefully at both programs, we will notice that both programs need to execute practically the same set of functional tests. The only difference is that program 2 has included tests related to the memory

functionality, adding 20 more test cases to the set of tests. Considering that program 1 has a total of 65 test cases and program 2 has a total of 85 test cases, we can conclude that program 2 has 76% redundant tests after testing program 1.

Table 1.1: Test Cases of the Calculator SPL

	Features	Test Cases	Program 1	Program 2
Core	Exit	5	X	X
	CE	5	X	X
	C	5	X	X
	Backspace	5	X	X
Operations	Add	5	X	X
	Sub	5	X	X
	Div	5	X	X
	Mul	5	X	X
	Signals	5	X	X
	Square Root	5	X	X
	Percentage	5	X	X
	Reciprocal	5	X	X
Language	Chinese	5		
	English	5	X	X
	Spanish	5		
Memory	M+	5		X
	MS	5		X
	MC	5		X
	MR	5		X
	Total number of tests:		65	85

As soon as we generate more calculator programs that each need to be tested, the amount of redundant testing will rapidly increase since all programs belong to the same family of products and share core functionality. For an overall view, consider testing all products from the calculator SPL. Considering that we need to test each of the 144 products individually, we will have a total of 12240 tests of which 97.91% test cases can be classified as redundant. If we can leverage this information when testing subsequent products from the same product line we may be able to avoiding running the redundant tests on each new product and save testing time. While previous

research on software testing for product lines has focused on *test case reuse* which can be defined as reusing individual test cases (or assets) between products there has been little research on this issue of *redundancy* between products. We highlight our main contributions with respect to this problem next.

1.2 Contribution

In this thesis we focus on the issue of reuse of test cases between products in the software product line. We have four primary contributions that we make to the SPL testing community.

First we define testability for software product lines in terms of test case reuse across products and develop key metrics to measure the impact of different approaches for testing SPLs with testability in mind. We present these ideas in Chapter 3.

Second, we identify the various elements of a feature model that we believe will lead to a reduction in the number of test cases that must be run across the entire family of products. We provide examples and insights into why the feature model itself may be an indicator of how testable an SPL is. This has the potential to lead to *design for testability* in early stages of SPL development; e.g. during domain engineering. This work is presented in Chapter 3.

Third we propose a black box approach for testing software product lines that takes into consideration types of variability of feature models identified previously. We hypothesize that our approach can reduce testing effort while retaining good fault detection in the presence all kinds of variability. Our approach is called the FIG Basis Path method. This is presented in Chapter 4.

Finally, we report results of an experiment performed with two software product lines to evaluate our new testing approach. The results show that we can achieve the

same fault detection results using our method as we can if we test all products. We do this using as few as, or fewer products than two other alternative approaches. The results of our case study can be found in Chapter 5.

Chapter 2

Background and Related Work

This chapter provides background on concepts required to understand this research as well as related work. Section 2.1 discusses software testing, including topics such as control flow graphs and basis path testing. Section 2.2 describes software product lines. Section 2.3 discusses testing for software product lines.

2.1 Software Testing

Software testing is an important phase of the software development life cycle. The main goal of software testing is ensure that the software satisfies the system requirements as well as reveals faults that may exist in the system. Software testing must be done efficiently and effectively [12].

Software testing strategies can be divided into functional (black-box) or structural (white-box) testing. A black-box strategy uses the requirements of the system to create tests without any knowledge of the underlying program code and/or structure, while the white-box strategy uses the control flow and data structure of the program to generate tests. Besides these two strategies, one can use a gray-box approach

to testing which is a combination of white-box and black-box techniques. Software testing strategies can be applied to different phases of the software development life cycle which includes unit testing, integration testing, system testing and others. *Unit testing* is the lowest level of testing in which individual units of the source code are tested by developers from a white box perspective. It validates a function or procedure of the system and requires detailed knowledge of the program design and code. As the system grows, individual software modules are grouped, and then tested together. This phase of testing is called *integration testing*. *System testing* performs a complete and integrated testing evaluation of the system requirements covering all parts of a system. Integration and system testing typically use a black box testing approach and are often performed by test teams that are separate from the module developers.

2.1.1 Black-box Techniques

Black-box testing, also called functional testing and behavioral testing, is a technique that uses system requirements to create test cases for a system [5]. This strategy can ensure that all the system requirements were implemented, but cannot guarantee that all paths in the system are executed since it has no knowledge of the internal structure of the program.

The typical black box test design strategies are equivalence partitioning [54], boundary value analysis [51], decision table testing [5], pairwise testing (Combinatorial Testing) [14] and state transition tables. Equivalence partitioning and pairwise testing use heuristics to select the input combinations, while state transition tables use heuristics to select paths from a behavioral model.

The category partition method [54] is a specification based method that parti-

tions the system inputs into parameters and categories of environment factors. The software engineer identifies functional requirements that can be tested separately and classifies them into categories. Next, a set of mutually exclusive choices and constraints are attributed to each of the categories. This information is used to create a test specification for the system, written in a test specification language, known as "TSL".

Combinatorial testing is a technique where all t-way combinations of values of a parameter model are exercised through test cases. The base object used to select the sample is a mixed level covering array. A mixed level covering array, *MCA (N, t, k* $(v_1, v_2, \ldots v_k)$*)*, is an $N \times k$ array. The k columns of this array are called factors, where each factor, f_i, contains v_i symbols. When more than one factor shares the same value for v, we can use a superscript shorthand notation to indicate how many consecutive factors have that value. The sum of superscripts in this notation is equal to k. For example, an (MCA(N, t, 2^3, 3^1, 4^2)) has six factors. The first three have two values, the fourth one has three values and the last two have four values. In an MCA all ordered t-way combinations between factors are covered at least once. Using a pair wise approach, we can guarantee that all pairs of values between factors are included in at least one product. A mixed level covering array will also be called simply a covering array in this thesis. The covering array has been shown to be effective in testing many types of configurable software [59]. For more details, refer to [14–16, 44].

2.1.2 White-box Techniques

White-box testing, also called structural testing, is a technique that uses the data structure of the program to create the test cases for the system [5]. The main goal of this technique is to provide a set of inputs that will cover some specific criteria, testing

the intended and unintended software behavior. This strategy cannot ensure that all the system requirements were implemented, but can guarantee that a specified set of code elements in the system are executed.

The typical white box test design techniques are control flow testing, data flow testing, branch testing and path testing. Next, we detail two of the white box techniques: control flow testing and path testing.

Control Flow Testing

Control flow testing is a white-box technique that makes references to the control flow graph (CFG) of a program [32]. The control flow graph is a directed graph that consists of a set of nodes and edges where nodes represent a statement or predicate in the program, and edges represent the flow of control between nodes. A predicate node is a node containing a condition. Usually, edges originating from a predicate node are labeled. Each CFG contains a unique entry node and a unique exit node as well as predicate values connected by edges forming paths of sequential nodes between the entry and the exit node. Figure 2.1 presents the graphical representation of the basic control flow graph structures.

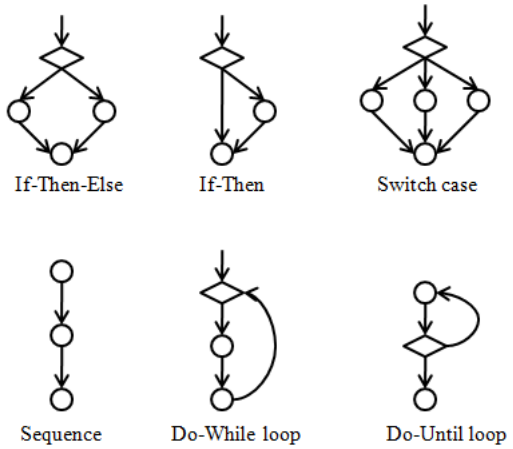

Figure 2.1: Control flow graph notation

There are numerous ways to create a control flow graph from a program. A CFG can be created manually or automatically. In Java, for example, a CFG can be constructed from the Java source code, or from the Java bytecode.

Based on the control flow graph of the program, test cases are created with the purpose of creating an adequate set of test cases. The adequacy of the test cases can be measured by selection criteria, such as statement coverage, branch coverage, and predicate coverage.

Require: m and n are both greater than zero
m $\geq$ n for efficiency

```
    euclid (int m, int n)
 1: int r;
 2: if (n > m) then
 3:     r = m;
 4:     m = n;
 5:     n = r;
 6: end if
 7: r = m % n;
 8: while (r != 0) do
 9:     m = n;
10:     n = r;
11:     r = m % n;
12: end while
13: return  n;
```

Figure 2.2 shows an example taken from [74]. This example computes the greatest common divisor (GCD) of two numbers; the example contains two of the basic common structures of a program, if and while.

Basis Path Testing

Basis Path Testing is a white box technique defined by McCabe [43] that uses the control flow structure of the program to design test cases. Based on the cyclomatic complexity measure, the basis path technique creates a basis set of paths for any graph.

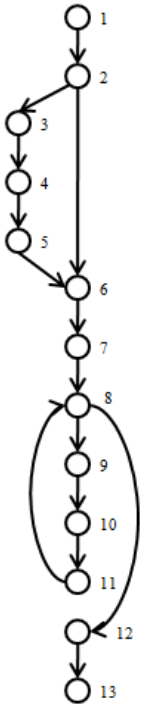

Figure 2.2: Greatest common divisor control flow graph

The cyclomatic complexity measure is an important property of the basis path technique. The cyclomatic complexity measure not only defines the maximum number of linearly independent paths in the graph but also ensures that every edge of a flow graph is traversed by at least one path in every basis set. If the graph has a loop, the Basis Path method restricts the edge traversal in order to avoid infinite loops and minimize the realizable complexity [74]. The cyclomatic complexity can be calculated by the following formula:

$$v(G) = e - n + p,$$

where G is a graph with n vertices, e edges, and p connected components.

Besides facilitating the creation of test cases for the program, the technique can also help in the test planning process and assessing the testability of the program.

The basis path testing technique is composed of four steps:

1. Represent the control flow graph of the program
2. Calculate the cyclomatic complexity
3. Choose a basis set of paths
4. Create test cases for each path.

Table 2.1 shows the basis set of paths for the GCD example introduced previously in Figure 2.2. Since the cyclomatic complexity of this graph is three, there is a basis path set of 3 paths.

Table 2.1: Basis Set of Paths of Euclids' GCD Program

	Paths
1	1 2 6 7 8 12 13
2	1 2 3 4 5 6 7 8 12 13
3	1 2 6 7 8 9 10 11 12 13

According to Watson et al. [74], basis path testing can also be applied at the module level aiding testing of integrated components. To use basis path testing at the module level, a design reduction technique in the control flow graph is essential since all the control structures that are not involved with module calls need to be removed.

The basis path technique, at the module level, helps to reduce the number of integration tests since it focuses only on components that have decision points (calls) to others and the design complexity can be significantly less than cyclomatic complexity.

2.2 Software Product Lines

Software product line engineering (SPLE) has been shown to be a very successful approach to software development that allows engineers to build families of products

that share some functionalities in short time and with high quality [70]. This paradigm has received attention in industry and the software research community as it shows how the development of products can be improved and more importantly how to respond quickly and effectively to market opportunities.

According to [12], a software product line is "a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular segment or mission and that are developed from a common set of core assets in a prescribed way."

The SPL development approach consists of two main phases: development for reuse called domain engineering and development with reuse called application engineering (See figure 2.3). Domain engineering is the phase where the variability and commonality of a system is identified, and a core set of reusable assets is developed. The assets include the requirements, design model, implementation, testing and other assets used in software development [56]. The key concept of the domain phase is the variability which is closely related to reuse. This results in the creation of a feature model as described below. Application engineering is the phase where an individual software product is built in accordance with the feature model.

Feature models are the artifacts most widely used to represent the variability of SPLs. In general, a feature model is a graphical representation that depicts the set of all features of an SPL and relationships among them. First proposed by Kang et al. [36], the feature model is a diagram where the common and the variable features, as well as the different relationships between them, are represented hierarchically. Lee et al. [41] review the concepts of features, and provide a guideline to create feature models in an efficient and effective way. Metzger et al. [47] provide a detailed survey of feature diagrams and the different representations found in the research literature. Benavides et al. [6] present a literature review of automated analysis of feature models.

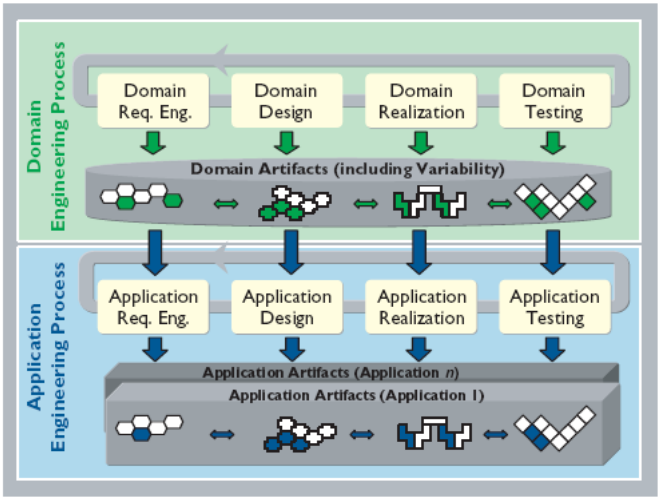

Figure 2.3: Framework from SPL engineering [57]

Next, we describe the notation of the basic features of a feature model.

- Mandatory. A parent feature has the mandatory relationship when all its child features must be included in all products of the SPL. In our calculator SPL example, the mandatory feature groups are Core, Operations, and Language. The Core feature group has two mandatory child features, Exit and Clear. The Operations feature group has four mandatory child features: Add, Sub, Div and Mul. The Language feature group is an alternative feature, as explained bellow.

- Optional. A parent feature has the optional relationship when any number of its child features can be included in among the products derived from the SPL. In our calculator SPL example, the optional features are Backspace, Signal, Reciprocal, Square Root, Percentage and the memory set of functionalities. Those functionalities may not be present in some products.

- Alternative. A parent feature has the Alternative relationship when only one child feature can be included in the products derived from the SPL. Language is

the alternative feature of our example, the calculator SPL. Calculators derived from this SPL must contain only one language.

- Or. A parent feature has the Or relationship when at least one of the child features must be included in the products derived from the SPL. In our calculator SPL example, the Or features are the memory store functionality. Products may contain one of the functionalities, M+ and MS, or both can be included.

In addition to the parental relationship between features, the feature models also contain cross-tree constraints between the features. The constraints are:

- Requires. The Requires constraint represents an implication relationship between features. If a feature A requires a feature B, every product that includes feature A must also include the feature B. In our calculator SPL example, the memory recall feature requires the store feature. This means that a selection of memory recall feature is only meaningful in connection with the Store feature.

- Excludes. The Excludes constraint represents a mutual exclusion relationship between features. If a feature A excludes a feature B, both features cannot be present in the same product.

Some extensions of the feature model have been proposed in the literature. Batory [4] has proposed constraints in the form of generic propositional formulas; Some authors [19, 63] have proposed extensions with UML-like multiplicities of the form [n,m] with n being the lower bound and m the upper bound; Others [4] have proposed the inclusion of attributes to include additional information on the features. Feature models have also been used for generative programming [3, 18, 67], providing a model based approach to the realization of product lines.

There has been a lot of work on feature modeling [2, 3, 36, 56, 64] of which we have presented only a small subset.

2.2.1 Orthogonal Variability Model

The Orthogonal Variability Model (OVM) [56] supports the abstract and consistent representation of software product line variability. In this model, the core concepts are the variation point (VP) and the variant (V). Each variation point has at least one variant, and the association between these elements describes their dependency.

In OVM, the dependencies between features can be Mandatory, Optional and Alternative. For alternative dependency, a group of variants is associated with a cardinality [min..max] which determines the total number of variants selectable for one product family. The authors also include the term of "variability constraint" to represent the constraints in the model. OVM also graphically represents the constraints between features. A constraint may be defined in three different ways: (1). between variants (Vs) and variants (Vs); (2). Between variants (Vs) and variation points (VPs); and (3). Between variation points (VPs) and variation points (VPs). Figure 2.4 shows the graphical notation for OVM modeled with the OVM Improver available at [55]. The OVM Improver tool supports the modelling of variability including the detection and correction of defects.

To illustrate this notation, we model our calculator SPL example in OVM (Figure 2.5). The calculator SPL has five variation points (VPs): Core, Operations, Language, Memory and Store. Each VP has an association with at least one variant. The Core VP has three mandatory variants, Exit, Clear (C), Clear (CE) and one optional variant, Backspace. The Operations VP has eight variants which four are mandatory (Add, Sub, Div and Mul) and four are optional (Signals, Square root,

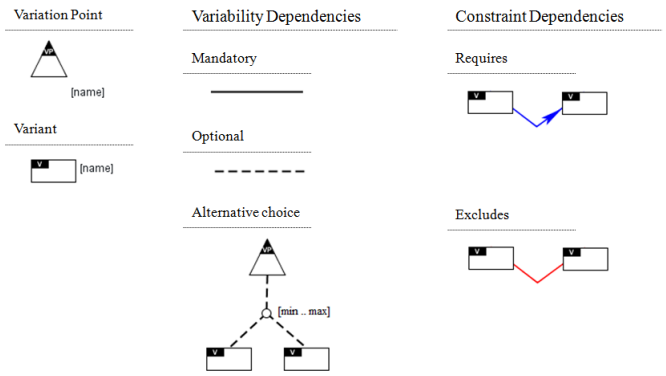

Figure 2.4: OVM notation

Percentage and Reciprocal). The Language VP has an alternative dependency with cardinality of 1 to 1. The variants for the Language VP are Chinese, English, and Spanish. The Store VP also has an alternative dependency, but with a different cardinality, 1 to 2. The variants for the Store VP are M+ and MS. The require constraint is represented between variants recall and store, and between variants clear and store.

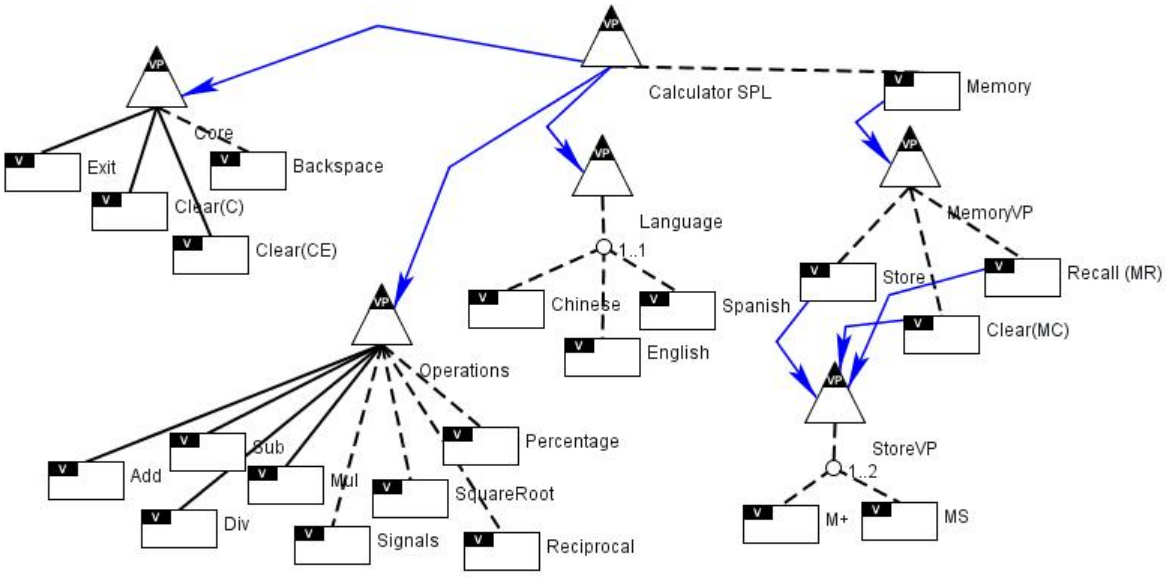

Figure 2.5: Calculator SPL feature model

2.3 Testing Software Product Lines

Software product line development promises to develop a family of products in short time with high quality at lower costs [70]. To achieve this goal, quality assurance became an essential part of the development process. Quality attributes such as correctness and reliability have begun to receive attention from industry and the research community as a consequence of the efforts to use more effectively the assets of an SPL throughout the products [25, 49].

Most of the research done for software product line development is focused on requirements, design, and implementation. Pohl et al. [57] describe six essential principles for SPL system testing that should be taken into consideration when developing testing techniques for SPL engineering. Several testing challenges for SPLs are identified in [50,66] of which the main one is the need to test many closely related products that all are part of a single specification. Jaring et al. [34] claims testing in software product lines does not always optimally benefit from reuse. Kolb and Muthig [38] point out, however that testing has not made the same advances as other parts of the SPL lifecycle and remains a bottleneck in SPL development.

Jaring et al. [33] 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 [45] introduces a set of testing techniques for software product lines including core assets of testing. These techniques are similar to techniques used in software development of single systems and does not address reusable test specifications. McGregor [44] and Cohen et al. [15] have suggested ways to reduce the combinatorial space by sampling products for testing using combinatorial interaction testing [13]. We use this technique as a comparison method in our case study.

Several authors [7, 52, 53, 62] have proposed the use of use cases to systematically reuse test specifications. Olimpiew et al. [53] introduces CADeT (Customizable Activity diagrams, Decision tables and Test specifications). CADeT is a functional test design method that utilizes feature-based test coverage criteria and use cases creating a reusable set of test specifications. Nebut et al. [52] uses use cases to generate product-specific functional test objectives, and a tool is proposed to automate the test case derivation. Reuys et al. [60, 62] presents ScenTED (Scenario-based Test case Derivation), a technique that supports the derivation of system test cases from domain requirements.

Other research on methods for testing families of products includes the PLUTO testing methodology [7], where the feature model is used to develop extended use cases, PLUCS (Product Line Use Cases), that contain variability which can formulate a full set of test cases using category partitioning for the family of products; however, this work does not provide methods for reducing testing effort across products.

Other work on testing software product lines includes that of Denger et al. [21] 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.

Feature models have been used to model the product space for instantiating products for testing [7, 15, 69]; for instance, the work of Uzuncaova et al. [69] transforms a feature model into an Alloy specification and uses this to generate test cases, while the work of Cohen et al. [15] uses feature models to define samples of products that should be included in testing. Schürr et al. [65] use a classification tree method for testing from feature models. Other extensions of feature models have been created for staged generation [17] or modeling constraints [20]. None of this work explicitly

uses feature models as we do, in a graph based representation that can be used to select products (and test cases) for testing as a control flow graph can be used for selecting paths and test cases. The work of Bachmeyer et al. [2] also uses a graph based representation of a feature model, but they do not use this in the testing process.

Chapter 3

Designing Software Product Lines for Testability

Software product line engineering (SPLE) promises to develop a family of products in a short time with high quality. The key to this is the reuse of artifacts. Reuse of software artifacts is the ability to leverage the use of the same artifact into another environment. Software product line engineering leverages reuse within all phases of the software life cycle development - from software requirement development through software maintenance. Software reuse can reduce not only the development costs, time, effort, and risk but also can improve the quality and productivity of the system. The most common type of reuse is the reuse of software components, however other artifacts can also be reused during software development, such as system architecture, design models, and others [39]. Jha et al. [35] claims that domain engineering is not the only key for reuse in SPLE, but that the documentation of software architecture supports the reuse and integration of reliable components.

Identified as a bottleneck in software product line engineering, *software testing* has become a crucial phase in the development cycle. Freedman [30] has argued that

design for testability is the most fundamental practice of software testing. Chanson et al [11] define software design for testability as a process of applying techniques and methods during the design phase in order to reduce the effort and cost in testing.

Software testability [8,9,28,71,72] is a subject that has been investigated over the years by both the research community and industry with the intent of making software testing more efficient and effective. According to Voas and Miller [72], software testing can reveal faults while testability cannot, but testability can suggest places where faults can hide from testing, something testing cannot do.

A lot of definitions have been proposed for software testability of single systems where each of the definitions reflects the testability of software from different points of view. In general, software testability is associated with the degree to which a software artifact facilitates testing of a program by reducing the testing effort. The IEEE standard glossary [22] defines testability as (1). "the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met" and (2). " the degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met".

Voas et al. [71, 72] define testability as a prediction of the probability of software failures when faults exist in the software. Freedman [28] analyzes observability and controllability software properties and introduces the domain testability concept. Binder [8] claims that software testability is a result of a combination of six factors: representation, implementation, built-in test, test suite, test support environment, and software process capability. In [9], the authors analyze two factors that influence the testability of a system: the number of test cases required to test the system, and the effort required to develop each of the test cases.

On the other hand, only a few researchers have defined testability for software

product lines. Jaring et al. [33] 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. Kolb and Muthig [38] discuss how to improve the testability of a product line architecture, and claim that the easiest case is testing products with no variability at all which means that some components are used by all products of the same family. The authors define testability of an SPL as "a characteristic of the architecture, design and implementation and with respect to the testing phase and an activity of the test process. Furthermore, product line testability refers to the testability regarding reusable product line components and the product line members built using them."

While all of the work just described aims at the core problem of software testability, none of if specifically considers testability of SPLs as an artifact of reuse in SPLs; something that can be determined by analyzing the feature model in the application engineering phase. In this work, we consider testability of SPLs from this perspective.

We begin with the conjecture that the testability of a software product line can be improved by designing the product line architecture (and resulting feature model) in a manner that supports reuse of testing assets across different products of the same family. Others have argued that variability decreases testability [38], 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 testability perspective and provide different opportunities to reduce testing effort, as we explain in Section 3.2.

By paying attention to testability early in the design phase, the testing phase can be improved by avoiding redundant tests and revealing faults. In the following sections, a detailed analysis of different types of features and their relationships in

the feature model is performed. We explore the issue of variability, and evaluate the correlation between the characteristics of each feature/relationship and testability.

We also analyze the different levels of granularity of features in the feature model and how each affects the testability of the SPL. This connection will help not only the traceability of test cases and features of the SPL, but the avoidance of redundant test cases as well. A cost per fault benefit may occur by selecting the appropriate granularity level of features.

3.1 Testability Measures

Before presenting the analysis of features and its correlation with testability, we present some metrics that can be used to evaluate the testability of feature models. We begin with metrics that are not derived directly from the feature model but that measure how testable a particular feature model is given a set of test cases that can be mapped to elements within the model. We leave a design-level set of metrics as future work.

In this section, we first introduce our definition of software product line testability, and then we develop key metrics to measure the impact of different approaches for testing SPLs. These key metrics are used for evaluating different characteristic of a feature model in Section 3.2.

3.1.1 Definition of Testability

Our definition for testability of software product lines is created from a family perspective. The testability of a software product line is the degree which feature models allow for greater reuse of test artifacts across products of the same family without losing effectiveness.

We consider the testability degree of a software product line to be high when it is possible to perform an efficient and effective test of all features, with a small set of products. Efficiency is the ability to execute a small set of test cases without avoiding redundant tests. Effectiveness is the ability to reveal faults within products.

3.1.2 Key Metrics

A software design leads to a system that must be tested, and that can be tested more or less efficiently and effectively. Several software metrics are based on internal characteristics of software to predict an expected behavior. We believe that there exists a way to measure the testability of an architecture/feature model itself by analyzing their characteristics.

As a first step we begin by defining metrics that indicate if a particular model is more or less testable based on reuse post-hoc. In future work we will develop metrics to measure testability on the architecture directly.

To introduce the testing efficiency and effectiveness in software product lines we introduce metrics for each: effectiveness and reduction.

The *effectiveness formula* is the ratio of the union of revealed faults for all the products to all revealed faults in the system. Given M products, we define F_i as the set of faults that are found when testing product i, where $1 \leq$ i $\leq$ M. If F_{SPL} is the set of faults found in all M products then we can define the effectiveness of testing a sub-set of products as:

$$Effectiveness = \frac{|\bigcup_1^M F_i|}{|F_{SPL}|} * 100 \tag{3.1}$$

For instance if we can find 10 unique faults when testing all products in the SPL and then find faults 1,2,3,5 in one product and faults 1,2,6,7 in another product our

effectiveness is 6/10 * 100 = 60%.

The *reduction formula* is the ratio of the total number of test cases used to test the SPL and the total number of test cases used to test all products of the SPL. Given M the number of products tested, we define T_i as the total number of test cases for product i, and N is the total number of products. Reduction is then defined as

$$Reduction = \left(1 - \frac{\sum_{i=1}^{M} T_i}{\sum_{i=1}^{N} T_i}\right) * 100 \tag{3.2}$$

For instance if we have a total of 10 products which each has 10 test cases, and we selected only 20 of all the test cases, we will have a reduction of (1 - 20/10*10) * 100 = 80%.

Given both the reduction and the effectiveness of testing for different feature models we can associate these with different levels of testability. For a full notion of testability we will need to tie this back to the architecture/design of the system itself.

3.2 Feature Model for Testability

Given our definition and metrics for testability we believe that different characteristics of the feature model lead to more/less testable SPLs. In this section, we evaluate the characteristics of variability and constraint relationships of a feature model and how they affect the testability of SPLs. We analyze the testability of feature models from a black box point of view using only requirements of the system without accessing the source code of the application. There are four types of variability: Mandatory, Optional, Or and Alternative and two types of relationship: Requires and Exclude. We not only analyze them, but also classify them into groups according to their testability. Table 3.1 summarizes the four variability types and two relationships analyzed

in this section providing a brief description of the variability and their testability classification.

Table 3.1: Classification of Features

Variability	**Description**	**Classification**
Mandatory	Features that *must* be present in all products.	Neutral
Optional	Features that *may* be present in all products.	Positive
Or	Features for which at *least one* variant *must* be selected over the possibilities.	Positive
Alternative	Features that are *mutually exclusive* and only one of the variants can be included in a product.	Negative
Constraint	**Description**	**Classification**
Requires	Constraint used between two features when one feature *requires* another feature's capability.	Positive
Excludes	Constraint used between two features when one feature *cannot be grouped* with the other.	Negative

Mandatory features are features that must be present in all products created from the SPLs. We also can see mandatory features as the commonality of all products created by an SPL model. From the testability point of view, we classify this group as neutral. Even though we classified this group as neutral, the testing of SPLs may be improved by testing the mandatory features only once avoiding the redundancy of reusing test cases over all the products, but we leave this as a future research direction.

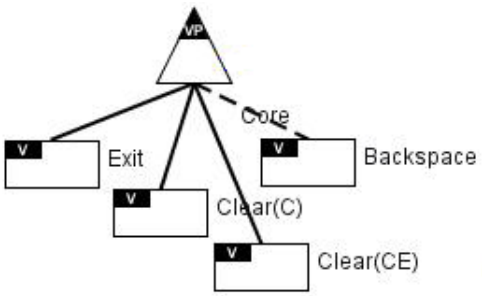

Figure 3.1: Mandatory features of the calculator SPL

For a better understanding, consider the Core variant point from our example, the calculator SPL. Figure 3.1 shows just this part of the OVM model. This VP specifies core functions of a calculator program. The Core VP has four variants of which three of them are mandatory. Since this VP is mandatory, each product will include at least the three mandatory variants. Even though all products contain the mandatory features, we can select the tests related to those variants to run only once and discard the related tests cases on the next products.

Optional features are features that may be present in products created by an SPL model and provide one way to represent the variability of an SPL. Optional features can be combined in many ways in an SPL model. These features can appear as a variation point, variant, and grouped together with other types of features (optional and/or mandatory). From the testability point of view, we classify this group as positive since it is possible to include all the optional features, and valid features, in one product having the largest number of features activated. Since reuse is the key concept for SPL, we can exhaustively test the feature only once, and discard it from other instances.

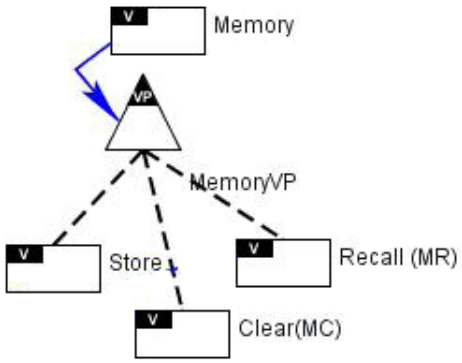

Figure 3.2: Optional features of the calculator SPL

For a better understanding, consider the memory variant from our example, calculator SPL. Figure 3.2 shows just this part of the OVM model. This VP specifies

the memory capability of a calculator program. Since this variant is optional, not all products of this SPL will include this feature. We improve the reuse of this feature by exhaustively testing the product that contains the memory feature, and discard the tests of this feature in other products that may contains the feature.

Or features are features for which at least one variant must be selected over the possibilities. For these the product will have one or more variants under the same variant point. The characteristics of this feature are highly correlated with the Optional feature group since they have practically the same behavior. We classify this group as positive. Like the Optional group, the Or feature group allows us to include all possible variants in one product.

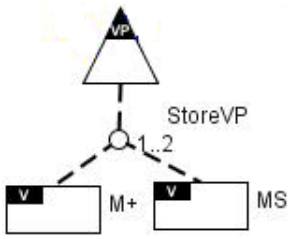

Figure 3.3: Or features of the calculator SPL

For a better understanding, consider the Store variant point from our example, the calculator SPL. Figure 3.3 shows just this part of the OVM model. This VP specifies how the numbers will be saved in the memory of a calculator program. The store VP has two variants: M+ and MS and has a cardinality of 1 to 2. Since this VP is optional within the SPL, we can include both variants (M+ and MS) into the same product, and test it exhaustively. There is no need to test it again on other products of the same family.

Alternative features are features that are mutually exclusive and only one of the variants can be included in a product. From the testability point of view, we classify this group as negative. To cover all the variants, we need n products where n is the

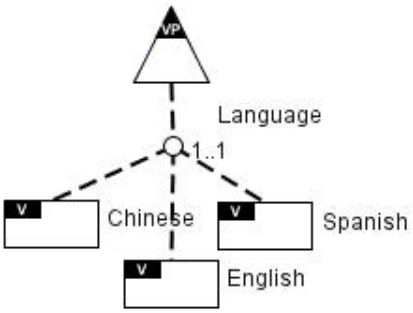

Figure 3.4: Alternative features of the calculator SPL

total number of variants under the alternative VP. In addition, the alternative features may further decrease the testability of SPLs when the behavior of each variant may affect the whole product, and consequently all other features (functionalities) need to be tested together.

For a better understanding, consider the Language variant point from our example, the calculator SPL. Figure 3.4 shows just this part of the OVM model. This VP specifies the language that will be used in menus, title, help and etc. The language VP has three variants: English, Spanish and Chinese. Since this VP is mandatory within the SPL, each of the languages must be tested together with the rest of the variants in the SPL. The VP can have only a single language included at a time, therefore, any testing that occurs with English and other variants such as with the optional operations (such as percentage) will need to be repeated with Spanish and Chinese. However, the optional operations may already have been tested with other parts of the SPL such as with the memory variants so there will be repetition in some of our test cases that cannot be avoided.

Now, consider the Store variant point. In our feature model, this variant point has the "Or" dependency but for illustrative purpose we will simulate the case in which this VP has an alternative dependency. Figure 3.3 shows just this part of the OVM model. This VP specifies how the numbers will be saved in the memory of

a calculator program. The store VP has two variants: M+ and MS. Since this VP is optional within the SPL, each of the variants may be tested only once avoiding the repetition of the tests cases. Controversially of language VP, the store variants (M+ and MS) have no impact on other variants of the calculator SPL facilitating the discarding of this VP in testing other products.

Based on these characteristics, we can categorize the Alternative feature group into two smaller groups: Alternative Features that affect the whole SPL and Alternative feature that affect only their own functionally. Unfortunately, the decision of which subgroup this feature belongs to must be made by the Software Engineer team.

The **Requires Constraint** is a constraint between two features, and it is used when one feature requires another feature's capability. When one of them is selected for a product, the other must also be present in the same product. From the testability point of view, we classify this group as positive since it is possible to include the "required and requires" variant in the same product. Based on this characteristic, we suggest that we always include the required and requires variants in the same product. This procedure will avoid the redundancy in testing of variants that are required by another variant.

For a better understanding, consider the Memory variation point which has three optional variants: Store, Clear and Recall. Figure 3.5 shows just this part of the OVM model. Even though all the variants have different behaviors, the Clear and Recall variants need the Store variant capability to work properly. Based on this, we suggest selecting Recall and Clear variants every time that the Store variant is selected to become part of one product.

The **Excludes constraint** is a constraint between two features, and it is used when one feature cannot be grouped with the other. This means that when one of them is selected for a product, the other feature must not be present in the same

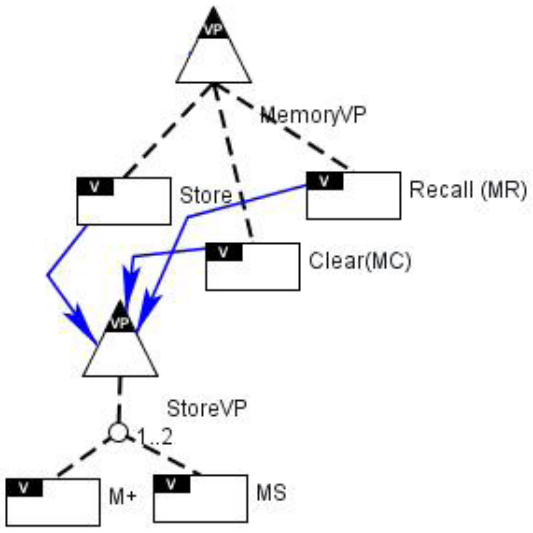

Figure 3.5: Requires constraints of the calculator SPL

product. From the testability point of view, we classify this group as negative since we will need two products to test both features.

3.3 Granularity of Feature Models

Different phases of life cycle development may require feature models of different granularity. For example, fine-grained feature models are essential when extracting features from legacy applications, while a coarse-grained feature models is enough at the requirement level. Determining the appropriate level of granularity for features in a feature model is still an open question.

In the previous section we extended the definition of testability for SPLs. In this section we analyze different levels of granularity of features in the feature model and evaluate how each of them affects the testability of an SPL considering the relationship between the two properties of granularity and traceability of a program.

3.3.1 Granularity

Granularity is the degree to which a system can be broken down into it is behavioral entities. In feature models, features can have different levels of granularity. A feature can represent a whole functionality, an extension of functionality, a part of functionality, or even a few lines in the program.

Granularity of features in feature models is a topic of feature modeling that has received a little attention by both the research community and industry. Kästner et al. [37] discuss the effects of granularity of features for two common ways to implement an SPL: the compositional approach and the annotative approach. The authors point out that a compositional approach does not support fine-grained extensions of feature models while an annotative approach does even though it introduces readability problems. Lee et al. [41] provide guidelines for refinement of features in the feature model. The authors argue that features must be easily mapped to architectural components, and this will enhance the traceability between architectural components and features.

As SPLs grow in size and complexity designing the right granularity of feature is required for feature models. If granularity is too fine, then feature models can become complicated, unreadable and unmaintainable [37]. If granularity is too coarse, then feature models can hide some important behavior of an SPL losing the connection with other models such as the architectural model.

For a better understanding, consider an extension of our example, the calculator SPL. This extension includes two kinds of number systems: decimal and hexadecimal. Hexadecimal is a positional numeral system with base 16. It uses sixteen distinct symbols, 0-9 to represent values zero to nine, and A, B, C, D, E, F to represent values ten to fifteen. A few features of the calculator SPL are affected by this new

addition where each operation of the calculator SPL needs to be performed with two types of number systems.

We can model this new addition in two different ways. Figure 3.6 shows the coarse-grained version of the feature model, while Figure 3.7 shows the fine-grained version. As we can see, both models use different granularities to model the new addition previously introduced. Figure 3.6 uses a coarse granularity by adding only a new variation point called Number Systems and associates it with the calculator SPL by adding a requires constraint. Figure 3.7 used a fine granularity adding a new variation point Number Systems to each affected operation of the calculator SPL.

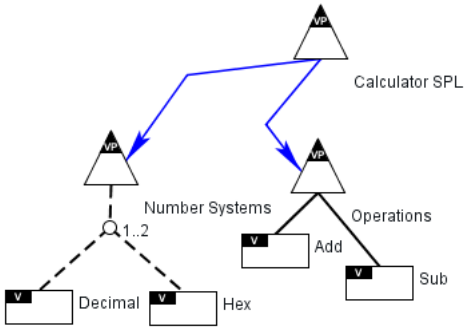

Figure 3.6: Coarse-grained feature model of the calculator SPL

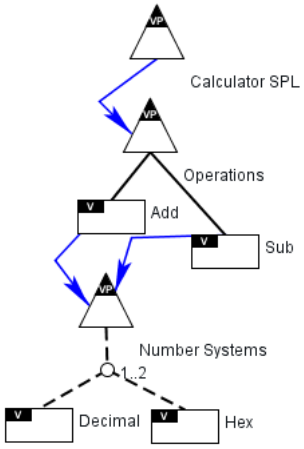

Figure 3.7: Fine-grained feature model of the calculator SPL

3.3.2 Testability and Granularity of Feature Models

Traceability is a technique that creates a relationship between system requirements and others artifacts of the system. It plays an important role in software development since it can verify that an implementation fulfills all the requirements and help with change management. For example, the traceability quality attribute supports software engineers in the identification of failed test cases (defects) with one specific requirement of the system.

For an illustrative example, suppose that we are testing the *Add operation* within a product that contains the two types of number systems previously introduced, and a failure happened. With the coarse-grained feature model (Figure 3.6) we can identify that there is a problem with the *Add* feature but we are not able to identify which number system failed since the add operation has a relation with each type of number system. However, if we consider the fine-grained feature model (Figure 3.7) we can easily identify which number system failed on the execution of the *Add* operation. The fine-grained feature model made it easy to establish the connection between the operation performed and the number system used.

We believe that a proper connection must exist between test cases and features. This connection will help not only the traceability of test cases and features of the SPL, but the avoidance of redundant test cases as well. The right level of granularity can bring several advantages when considering testability of software product lines. We enumerate some of these advantages here:

- Different levels of granularity will produce differing numbers of test cases and different amounts of re-use. By selecting the appropriate granularity testing the SPL may become both more efficient and more effective.

- Traceability between features and test cases can improve black box testing.

Choosing the correct granularity can improve traceability.

- Having the correct level of granularity will have an impact on the number of test cases that may be re-used between different SPL products improving re-use and avoiding redundancy.

Given the possible impact of granularity on testability we believe that this should be part of our model. In Chapter 5 we examine granularity as part of our case study to understand more about this issue. But we leave much of the work of defining the correct granularity for SPL testability as future work.

Chapter 4

Leveraging Redundancy for Testing via Feature Models

In the previous chapter, we defined testability for software product lines in terms of test case reuse across products and developed key metrics to measure the impact of different approaches for testing SPLs with testability in mind.

We now propose a new black box approach for testing software product lines that takes into consideration types of variability of feature models identified in the previous sections. We hypothesize that our approach can reduce testing effort while retaining good fault detection in the presence of the most types of variability.

Our methodology, which we call the FIG Basis Path method, involves five main steps as introduced in Figure 4.1. In step 1, we associate each feature of the feature model with the use cases requirements for the software product line. In step 2, we create test cases that cover all scenarios of each use case. In step 3, we transform the feature model into a *feature inclusion graph*; that is, in essence, a feature model dependence structure of an SPL. In step 4, we walk the feature inclusion graph to generate a subset of independent paths that cover the graph for testing where each

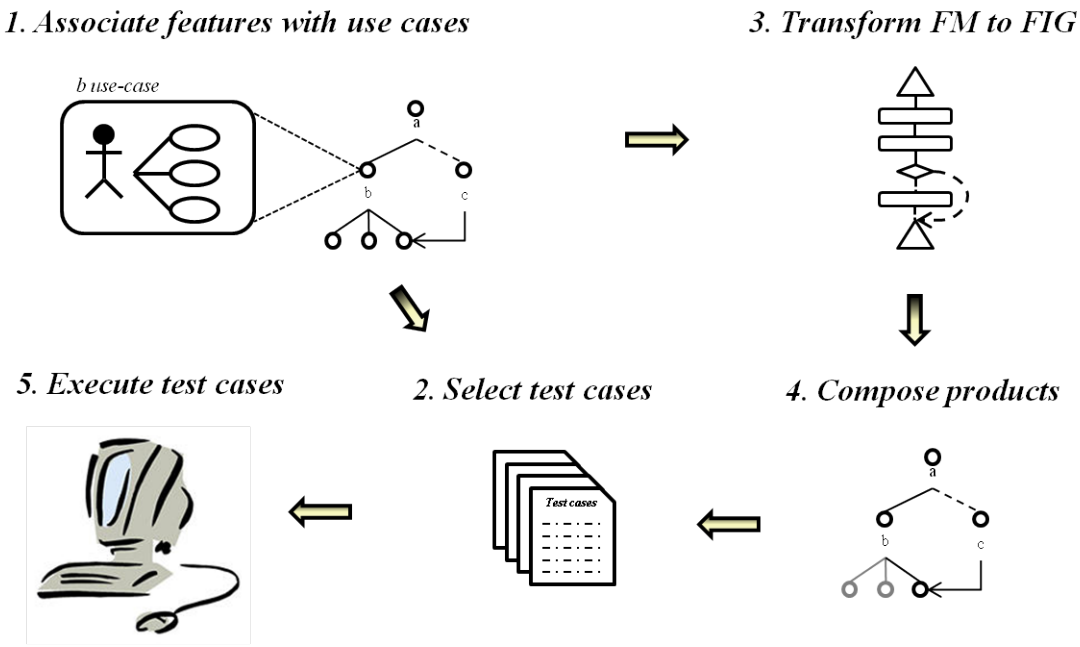

Figure 4.1: FIG testing framework

independent path can be seen as a valid product of the software product line. Steps 3 and 4 may happen in parallel with step 2. Finally, in step 5, we select all test cases related with each feature of the product and execute them.

Next, we described each step of our methodology in detail.

4.1 Associate Features with Use Cases

Use case modeling offers poor assistance in modeling variability of software product lines. A lot of work has been suggested by the community to cover this deficiency [23, 26, 40]. Eriksson et al. [23, 24] argue that association of features and use cases is a many-to-many relationship where one feature may be described by several use cases and one use case may be included in several features.

In this section, we describe the PLUSS approach. Essentially, the PLUSS approach maintains one use-case diagram for the SPL, and uses the feature model as a tool to

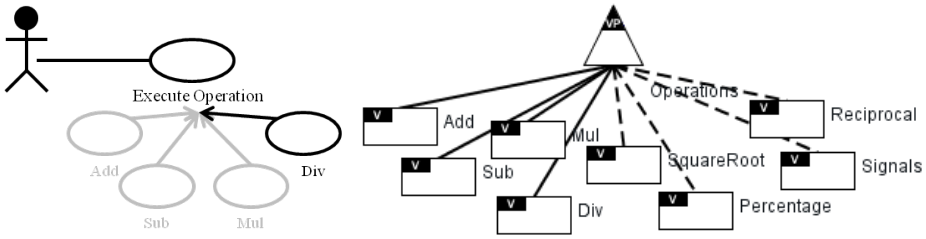

Figure 4.2: Operation use case and feature model

compose all the different products from the SPL. The authors created a meta-model that associates use cases, use-case realizations and features of the feature model. The meta-model is shown in Figure 4.3. For a detailed description, see [24].

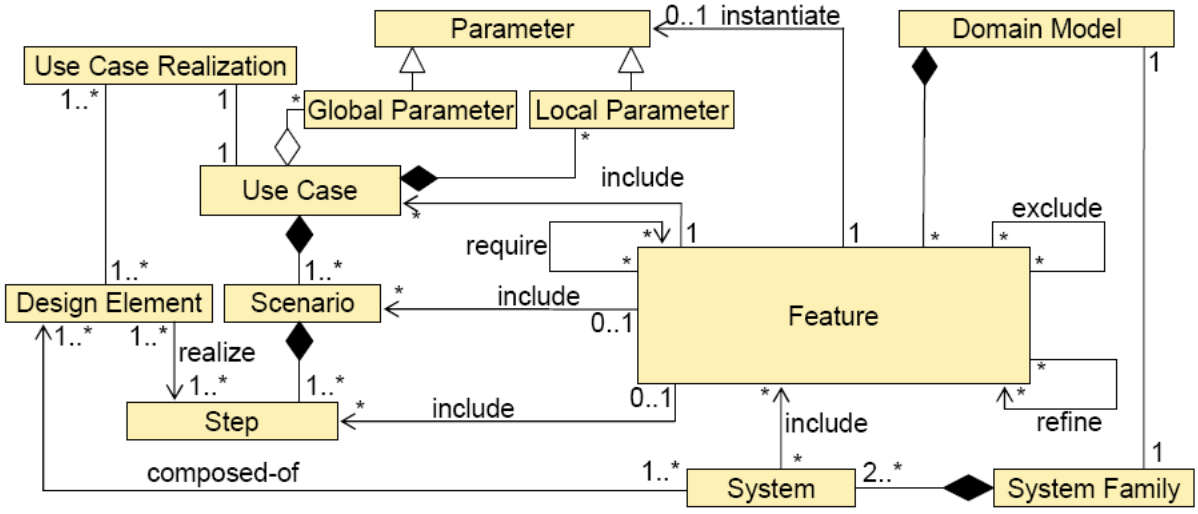

Figure 4.3: PLUSS meta-model [24]

For an illustrative example, consider the division operation use case for the calculator SPL example. The division operation use case describes how a division operation is performed by users of the calculator program. Figure 4.2 shows a snippet of the calculator feature model and the use case diagram. By applying the PLUSS meta-model, we associate the feature div on the feature model with the division operation use case.

In our methodology, we used the meta-model defined in PLUSS [24] to integrate use cases and features.

4.2 Create Test Cases

A large body of work on extraction of test cases from use cases has been done. Use cases [1, 68] are descriptions of a system's requirement using natural language that capture a system functional requirements. First, software engineers identify all scenarios (main and alternatives) of a specific use case. Scenarios are paths through flows of events in the use case. Then, test cases are created with the main purpose of covering all the scenarios thus identified.

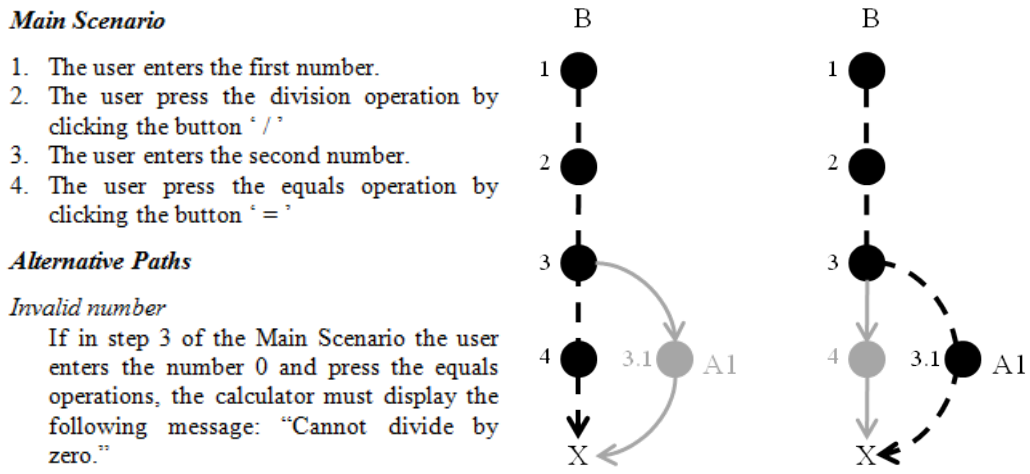

Figure 4.4: Division use case and paths

For an illustrative example, consider the use case for the division operation of our example, the calculator SPL. Figure 4.4 shows the Division operation use case. On the left side of the figure, we present the use case description of the division operation. On the right side, we present the graph representation of the use case. By analyzing the use case, we will create two test cases: one that executes the main scenario of

the use case, and another one for the alternative path. Table 4.1 shows the test cases created based on the division use case requirement and the steps covered by each of them. We highlight (using dashed lines in figure 4.4) the paths that we aim to execute by the test cases created for this use case.

Table 4.1: Covered Steps from Division Use Case

Test cases	Covered steps
1	1, 2, 3 and 4
2	1, 2, 3 and 3.1

4.3 Transform Feature Models into Feature Inclusion Graph

There has been a large body of work on feature modeling [2,36,64]. Some of this work includes transforming a feature model into a graph [2] - not for the explicit purpose of testing, but rather to incorporate more fine grained constraint information [20] or for generating the products themselves [3]. We do not know of other work, however, that transforms a feature model into a graph for the explicit purpose of test case selection.

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. We classify a testing technique based on the FIG as a black-box technique since the graph relies exclusively on the feature model of an SPL, a diagram created at the domain engineering phase. However, we take some advantage of white-box techniques, such as those that use control flow graphs and basis paths, to improve our testing methodology.

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 [31] 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 their dependencies, but we use a common method from control flow based testing to find a *basis path set* [75] 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 [56]. 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 of an SPL model (a variant can/cannot be selected for one specific instance). Figure 4.5 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 4.5) 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 (left in the 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). We show two optional features (again B and C) on the right side in Figure 4.5. Here in the figure (lower right) we have added three branched edges. The middle edge represents the case in which neither feature is included, while each of the other 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 do not handle the case in which multiple copies of a feature occur in our representation) 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.

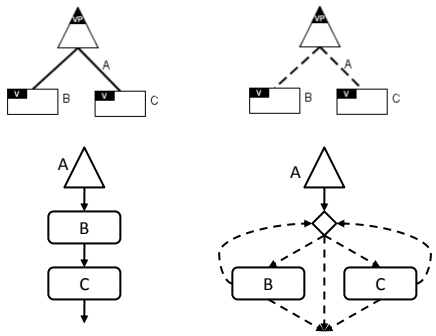

Figure 4.5: Mandatory and optional features

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. For a concrete example

we can apply this to the Core variation point and its variants in the calculator SPL. The Core variation point has three mandatory features (Exit, Clear (C) and Clear Entry (CE)) and one optional feature (Backspace) as shown in Figure 4.6. 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 (instead of the possible two instances in the SPL snippet).

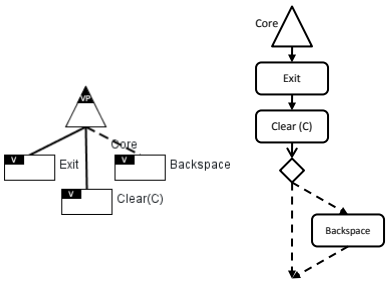

Figure 4.6: Calculator example

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 re-usability. Figure 4.7 shows three examples of alternative features in OVM (top) and their corresponding FIG (bottom). The first example (left) has cardinality 0...1, i.e., and is really an optional feature that can be included in 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.

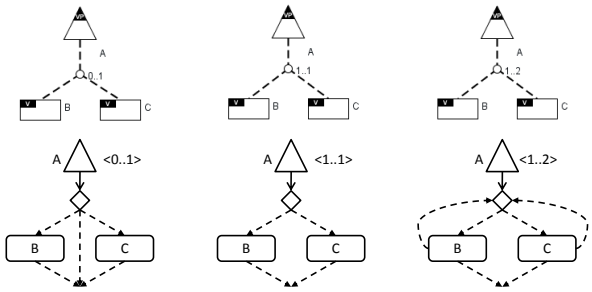

Figure 4.7: Alternative features

We see no reduction in paths.

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. Figure 4.8 presents our example, the calculator SPL, represented in the FIG notation.

4.4 Compose Products

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 method is our core algorithm called the *FIG Basis Path* method. 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 method called the *FIG Grouped Basis Path* method that tests subfamilies of the product line grouped by the alternative features in the SPL. We believe that this method will be incorporated into the development process more smoothly, where one

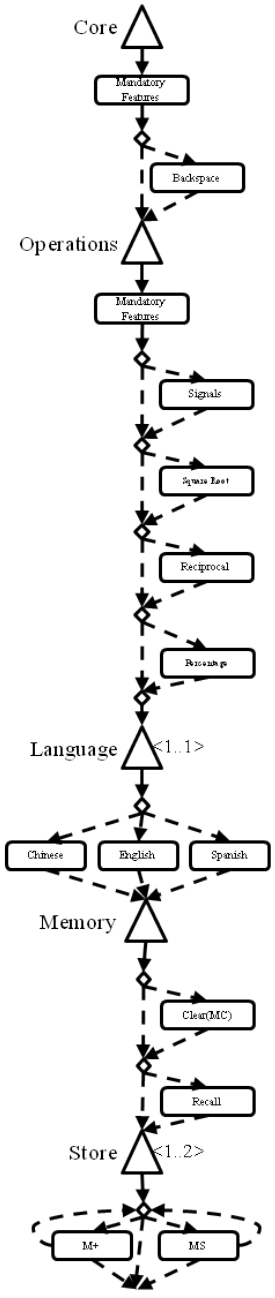

Figure 4.8: Calculator SPL Feature Inclusion Graph

particular subfamily is created at a time. The third method 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* method. The 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 [44] and Cohen et al. [15]. 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.

4.4.1 FIG Basis Path

The **FIG Basis Path** method (Algorithm 1) is based on the basis path algorithms in [75]. In this algorithm we assume that the FIG is built and that we have a set of constraints on the features. We begin by iterating through all paths in the FIG in a depth first traversal in order to ensure that we find the longest paths first (line 1). 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 (line 2). If it is, we then check if it is linearly independent with the other paths in BP (line 3). 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 (line 4).

Algorithm 1 FIG Basis Path Algorithm

Require: Set basis path (BP) empty.
BasisPath(**FIG**)
1: **for all** paths, P, ∈ FIG (using DFS order) **do**
2: **if** P is feasible **then**
3: **if** LinearlyIndependent(P, BP) **then**
4: add P to BP
5: **end if**
6: **end if**
7: **end for**

For example, suppose we want to select the minimum set of products in the calcu-

lator SPL. We show the FIG in figure 4.8. 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, 16 variants are selected, containing all of the mandatory features and optional features, one language - Spanish - and three variants from the Memory variant point: Store (M+), Recall (MR) and Clear (MC). The second path substitutes the Store variant from the previous path (M+) to MS. The third and fourth paths change only the Language variant. Table 4.2 lists all the paths created by this algorithm.

Table 4.2: Calculator SPL Instances Generated by FIG Basis Path

Product ID	Total Features	Features Selected
1	16	Exit, Clear, Clear Entry, Backspace, Add, Sub, Mul, Div, Square root, Percentage, Sign, Reciprocal, Spanish, Store (M+), Recall (MR), Clear (MC)
2	9	Exit, Clear, Clear Entry, Add, Sub, Mul, Div, Spanish, Store (MS)
3	8	Exit, Clear, Clear Entry, Add, Sub, Mul, Div, Chinese
4	8	Exit, Clear, Clear Entry, Add, Sub, Mul, Div, English

4.4.2 FIG Grouped Basis Path

The **FIG Grouped Basis Path** method (Algorithm 2) is a modification of the FIG Basis Path method, 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 in order to ensure that we find the longest paths first 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 each of these groups, we iterate through all paths. For each path, we check

if it is linearly independent with the other paths in BP (line 2). If it is independent we add it to BP (line 3).

Algorithm 2 FIG Grouped Basis Path Algorithm

Require: Set basis path (BP) empty.
Set FIG_GROUP with all feasible paths grouped by features.

GroupedBasisPathAlgorithm(**FIG_GROUP**)

```
1: for all paths, P, ∈ FIG_GROUP do
2:     if LinearlyIndependent(P, BP) then
3:         add P to BP
4:     end if
5: end for
```

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 in one group. All that contain Chinese go into in another, and the rest that contain English are put into another. In the calculator SPL all paths would contain at least one of these variation points so we do not have an extra group. 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. Table 4.3 lists all the paths created by this algorithm.

Table 4.3: Calculator SPL Instances Generated by FIG Grouped Basis Path

Product ID	Total Features	Features Selected
1	16	Exit, Clear, Clear Entry, Backspace, Add, Sub, Mul, Div, Square root, Percentage, Sign, Reciprocal, Spanish, Store (M+), Recall (MR), Clear (MC)
2	9	Exit, Clear, Clear Entry, Add, Sub, Mul, Div, Spanish, Store (MS)
3	16	Exit, Clear, Clear Entry, Backspace, Add, Sub, Mul, Div, Square root, Percentage, Sign, Reciprocal, Chinese, Store (M+), Recall (MR), Clear (MC)
4	9	Exit, Clear, Clear Entry, Add, Sub, Mul, Div, Chinese, Store (MS)
5	16	Exit, Clear, Clear Entry, Backspace, Add, Sub, Mul, Div, Square root, Percentage, Sign, Reciprocal, English, Store (M+), Recall (MR), Clear (MC)
6	9	Exit, Clear, Clear Entry, Add, Sub, Mul, Div, English, Store (MS)

4.4.3 All Features

The **All Features** method (Algorithm 3), does not require a FIG. This method 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 in one of three sets, Mandatory, Alternative and all others. We order features in descending order based on the number of constraints on that feature. We keep a set we call *used features* which starts as empty. This set contains all the visited features. 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 (lines 7-8) or violate an exclude constraint (lines 18-24), or are part of a requires constraint (lines 27-33)), until we have a product including the greatest number of unused features. We then update the used features and product sets (lines 25-26). We also include additional constraints to ensure that only one feature of an alternative variant point is selected (lines 34-36). Once all features have been included in at least one product we are done (line 40). The intuition behind this method is that we include all features in at least one product, but we pick products first that have the greatest number of new features, to minimize the size of the product space tested.

For the calculator SPL, we select three products that cover all features in the SPL. See Table 4.4. 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.

Algorithm 3 All Features Algorithm

```
Require: Set Mandatory with with all common variants.
         Set Alternative with a list of all alternative variants.
         Set allOthers with all variants of the feature model ordered by highly constrained.
         Set VPSet with a list of all variant point.
         Set excConstrainst with all pair-variants that has the excludes relationship.
         Set reqConstraint with pair-variants that has the requires relationship.
         Set usedFeatures empty.
   SelectInstance (v, G)
 1: for all VP ∈ allOthers do
 2:   for all v ∈ VP do
 3:     if (v ∈ VPSet) AND (v ∉ instance) then
 4:       instance = instance ∪ v
 5:       SelectInstance (v, G)
 6:     else
 7:       if v ∈ Mandatory then
 8:         add v
 9:       else if v ∈ Alternative then
10:         if any v of curVP ∈ instance then
11:           continue
12:         else if all v in curVP ∈ visited then
13:           instance = instance ∪ v
14:         end if
15:       else if v ∈ excludeSet then
16:         continue
17:       else
18:         if v ∈ excContrainst then
19:           vExc = getExcConstraint(v)
20:           if vExc ∈ usedFeatures then
21:             continue
22:           end if
23:           excluseSet = exclude ∪ vExc
24:         end if
25:         product = product ∪ v
26:         usedFeatures = usedFeatures ∪ v
27:         if v ∈ reqConstraint then
28:           vReq = getReqConstraint(v)
29:           for all vr ∈ VReq do
30:             product = product ∪ vr
31:             usedFeatures = usedFeatures ∪ vr
32:           end for
33:         end if
34:         if if parent(v) ∈ alternativeSet then
35:           continue
36:         end if
37:       end if
38:     end if
39:   end for
40:   Print product
41: end for
```

4.4.4 Covering Array

Our fourth method, the **Covering Array Method** is a testing technique that samples a set of instances in such a way that all t-way combinations are included [14,16].

Table 4.4: Calculator SPL Instances Generated by All Features

Product ID	Total Features	Features Selected
1	17	Recall, MS, M+, Clear, Chinese, Add, Sub, Mul, Div, Square-Root, Percentage, Reciprocal, Sign, Exit, Clear(CE), Clear(C), Backspace
2	9	Spanish, Add, Sub, Mul, Div, Exit, Clear(CE), Clear(C), Backspace
3	9	English, Add, Sub, Mul, Div, Exit, Clear(CE), Clear(C), Backspace

Table 4.5 lists the factors and values of the calculator SPL as a mixed level covering array (MCA). We can see that Language has a different size set of values than the other factors.

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.

Table 4.5: Calculator SPL Interaction Model

Product Line Options (factors)									
	Operations					Memory			
Backspace	Signal	%	Recip.	Sqrt	Language	M+	MS	MC	MR
On Off	On Off	On Off	On Off	On Off	English Spanish Chinese	On Off	On Off	On Off	On Off

For a better illustration of this method, consider the pair-wise testing of the calculator SPL. Table 4.5 shows a list of all possible features of this software product line that can be combined with each other. In this model, we have 10 factors: Backspace,

Signal, Percentage, Reciprocal, Square root, Language, M+, MS, MC and MR. Each of these factors has a set of possible values varying on the total number of values. The MCA for the calculator SPL is represented as MCA(2^5,3^1,2^4). Table 4.6 shows a set of 10 products that make up the calculator CA. This set of products represents all possible pairs of factor-2. For example, the pair of *(backspace, on)* and *(language, english)* appears in the third row; the pair of *(signal, on)* and *(MC, on)* appears in the second row. In this set of products, all pairs of features are covered in at least one product, but not all tuples. In this sample, we missed the combination of *(language, english)*, *(M+, on)* and *(MS, on)*.

Table 4.6: Testing 2-way Interactions from the Calculator SPL

		Operations					Memory			
	Backspace	Signal	%	Recip.	Sqrt	Lang.	M+	MS	MC	MR
1	Off	Off	On	Off	On	English	On	Off	On	On
2	Off	On	On	On	On	Spanish	On	Off	On	Off
3	On	On	On	On	On	English	On	On	Off	Off
4	On	On	Off	On	Off	Chinese	Off	On	Off	Off
5	On	Off	Off	Off	Off	Chinese	On	Off	On	On
6	On	Off	Off	Off	Off	Spanish	On	Off	Off	Off
7	Off	Off	On	On	Off	Spanish	Off	On	On	On
8	Off	On	On	On	On	Chinese	On	On	Off	On
9	On	Off	Off	Off	On	Spanish	Off	On	On	On
10	Off	On	Off	Off	Off	English	Off	On	Off	Off

4.5 Execute Test Cases

In this section, we present the last step of our methodology. At this point, we have created all the products that will be used to test the software product line. Each product is basically a composition of features. In this step, the software engineer creates a test suite for each of the products by selecting all the test cases that were associated with each feature (Step 2 of our method) included in the product.

For example, consider the execution of this step for one of the products selected by the FIG Basis Path method in Section 4.4.1. This product, called *p1* in this example, contains eight features: Exit, Clear, Clear Entry, Add, Sub, Mul, Div and English. Since we can apply the same procedure to all features, we will restrict this example with only one feature of program *p1*, Div feature. As we presented in Section 4.2, Div feature has two test cases that cover all the scenarios of the use cases. Table 4.7 lists the test cases for the Div features, the inputs and the expected output for each test case. These test cases are added into the test suite of program *p1* and executed by the test engineer. The test suite for program *p1* may be different when compared with others test suites for the calculator SPL since the test suite is made based on the features selected for the program. For example, all programs will contain the test cases described in Table 4.7 since Div is a mandatory feature of calculator SPL. However, programs that contain the memory functionalities included will have a different test suite than program *p1* that does not include the memory functionalities.

Table 4.7: Test Cases Selected for Div Feature of the Calculator SPL

Feature	Use Case	TC ID	1st number	2nd number	Expected Output
Div	Division Operation	1	9	3	3
Div	Division Operation	2	6	0	Cannot divided by zero.

Chapter 5

Case Study

To gain insights into the operation of the FIG Basis Path method we conducted a case study, comparing the approach to the three other approaches described in Chapter 4. Our goal is to address the following research questions:

RQ1: How does the FIG Basis Path method compare with other test methods?

RQ2: Can we reduce the effort required to test groups of products through the FIG Grouped Basis Path method?

RQ3: Can the feature model granularity impact the effectiveness of the FIG Basis Path method?

5.1 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 [42]; it is built using the AHEAD methodology and implemented as a series of .jak files [3] (an extension of the Java

language). The second SPL is a mobile phone software product line, named Mobile Media [27], created by Lancaster University and widely used in previous studies.

Table 5.1 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.

Table 5.1: 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

The Graph 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 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 Graph SPL feature model contains a total of 80 instances without constraints. After reading the documentation for the Graph SPL we created a feature model for it, as shown in Figure 5.1. To create this model we needed to resolve some ambiguity in the documentation and we also reduced the possible cardi-

nality for combinations for the variant point *Alg*. Ultimately we obtained 38 possible instances of the product.

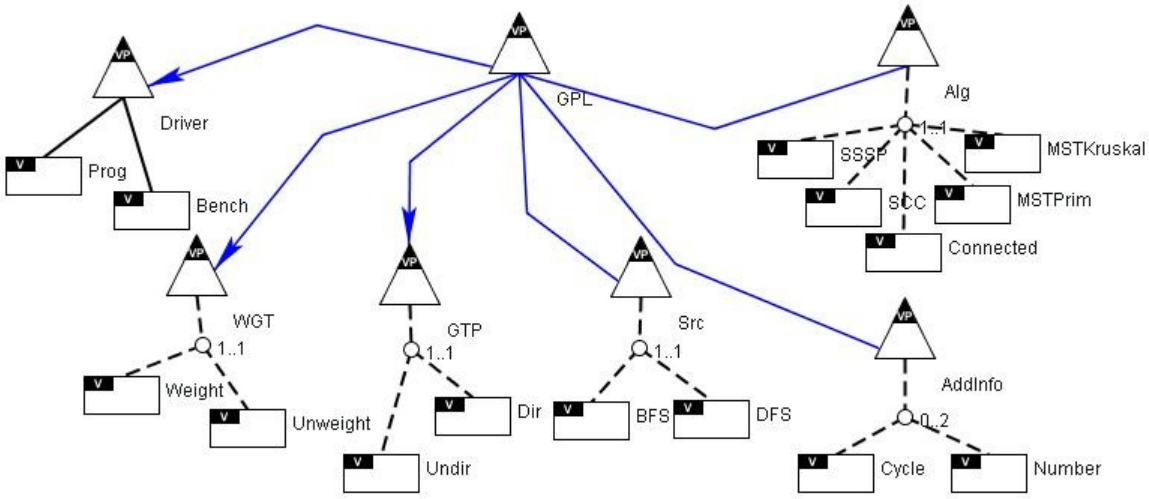

Figure 5.1: Graph SPL feature model

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. Until now, 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 five and six. In version five, 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 related to albums and labels. The version five of Mobile Media allows us to derive a total of 16 instances of the product. We present the feature model for version five in Figure 5.2.

Version six is a refactored version of version five and it includes one more type of media. In this version, users are allowed to manipulate two different types of media: photo and music. Both versions share a set of a few operations, as Album Management and Media Management, but they have different underlying code bases

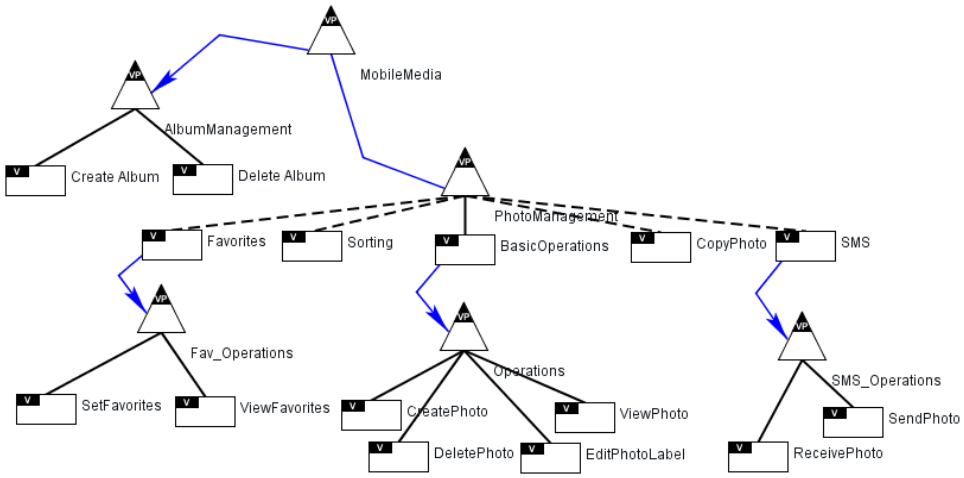

Figure 5.2: Mobile Media SPL feature model - version 5

due to the refactoring. The Mobile Media feature model allows us to derive a total of 24 instances of products. We present the feature model for version six in Figure 5.3. As we can see, new variation points, variants and constraint dependencies were included due to the addition of a new type of media.

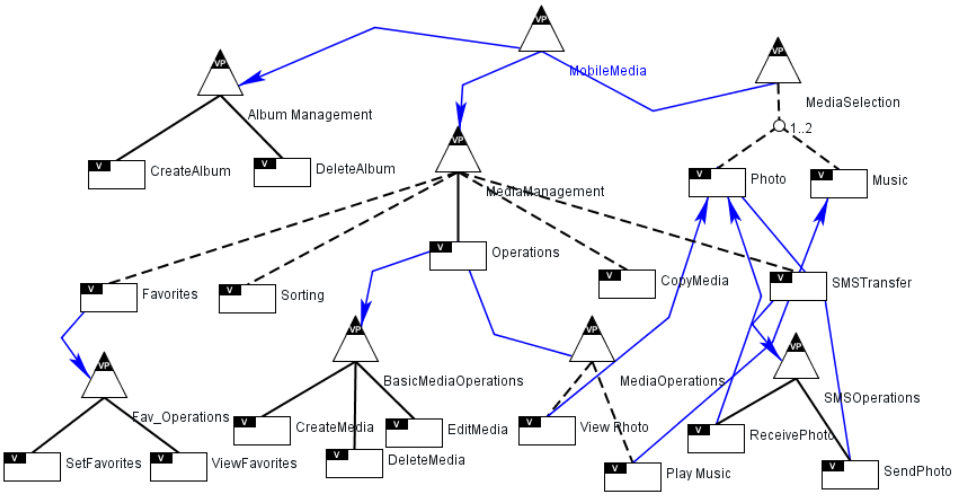

Figure 5.3: Mobile Media SPL feature model - version 6

5.2 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.

5.3 Test Suites

The test suites used in our study 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 as the alternative paths. 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 Graph SPL, the test suites are command line test cases. For the Mobile Media product line, the test cases are GUI based and implemented using a combination of two open source testing tools, Microemulator [48] and Abbot [73]. Microemulator provides a Java SE exclusive implementation of a mobile phone emulation platform with similar capabilities to the SUN wireless toolkit historically used for mobile phone application development. Abbot provides a means for automating user activities on the phone emulator thereby exercising the features of the MobileMedia subject. Abbot is implemented in Java and utilizes components of the Java AWT to generate event actions and collect screen images for comparison to an oracle. In Mobile Media version

five we had some non-deterministic test cases which we removed for our study. These ran deterministically for version six however so we left them in the test pool.

5.4 Fault Seeding

In the Mobile Media SPL, during the course of working with the system, we found ten actual faults in both versions 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 single *faulty versions*. We thus had ten faulty versions of this application for both version five and six. For the Graph SPL, 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 did not provide any information on the purpose of our study (and none had a complete running application to examine). We asked each student to seed ten faults into their set of files. This yielded 60 faulty versions of this application.

5.5 Results

In this section we examine the results of our study relative to our research questions. We begin with RQ1 which asks how the FIG Basis Path method compares with other methods. Table 5.2 shows the data for both SPL applications used in our study. 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 Graph SPL 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 FIG 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 (five products and 26 test cases); however, this technique missed three faults when compared with the other techniques.

Table 5.2: Required Test Cases and Faults Detected per Technique

	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
FIG 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
FIG 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 five, the All Features and FIG 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 six, the All Features and FIG 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.

RQ2 asks if we can reduce the effort required to test groups of products through the FIG Grouped Basis Path method. To answer it, we examine data shown in Table 5.3. This table shows the data grouped by the alternative features of each SPL. The left side of the table 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 FIG Grouped 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 Graph SPL, 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 5.3: Number of Test Cases Detected by Alternative Variants

Graph SPL						
	All Feasible Paths			FIG 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			FIG Grouped Basis Path		
Variant	#Product	#TC	#Faults	#Product	#TC	#Faults
Music	4	139	4	1	40	4
Photo	5	220	10	1	49	10

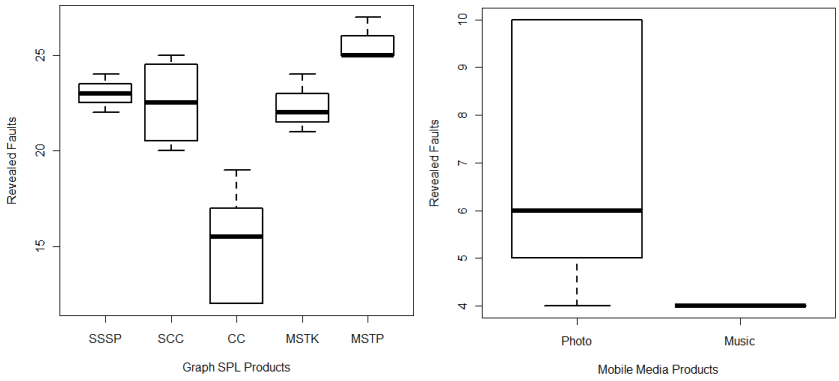

Figure 5.4: Number of test cases and faults detected grouped by alternative features

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 Graph SPL, 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 5.4). In each plot we see a range of fault detection, indicating that the FIG 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 FIG 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 belong to the same group. The data in Figure 5.4 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.

RQ3 asks if the feature model granularity can impact the effectiveness of the FIG Basis Path method. To answer it, we examine data shown in Table 5.4. This table shows the fault data grouped by the alternative features of Mobile Media SPL version

Table 5.4: Mobile Media Faults

Features		Faults			
		Photo		Music	
Name	Type	# faults	# faults revealed	# faults	# faults revealed
Create Album	Mandatory	0	0	0	0
Delete Album	Mandatory	1	1	1	1
Create Media	Mandatory	1	1	1	1
Edit Media	Mandatory	1	1	1	1
Delete Media	Mandatory	1	1	1	1
View Photo	Alternative	0	0	0	0
Play Music	Alternative	0	0	1	1
Set Favorites	Optional	0	0	0	0
View Favorites	Optional	0	0	0	0
Sorting	Optional	0	0	0	0
Copy Media	Optional	2	2	**2**	**0**
Send SMS	Optional	2	2	n/a	n/a
Receive SMS	Optional	2	2	n/a	n/a

six. The left side of the table shows the number of faults identified and the number of faults revealed by FIG Basis Path method for the Photo variant. The right side of the table shows the number of faults identified and the number of faults revealed by FIG Basis Path method for the Music variant. As we can see, the FIG Basis Path method reveals most of the faults but did not reveal the fault for copy music media. We discuss the implications of these results in the next section.

5.6 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 (Graph SPL) we see that the FIG 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 method was as effective at finding faults as all other methods, and cost 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. 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. A further discussion in faults of software product lines is provided in Section 5.7.

For RQ2, we see that it is possible to test parts of the product space more efficiently using the FIG Grouped Basis Path method when the feature model has alternative variant points. In the Graph SPL 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 FIG 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.

For RQ3, we see that there is a correlation between the granularity in the feature

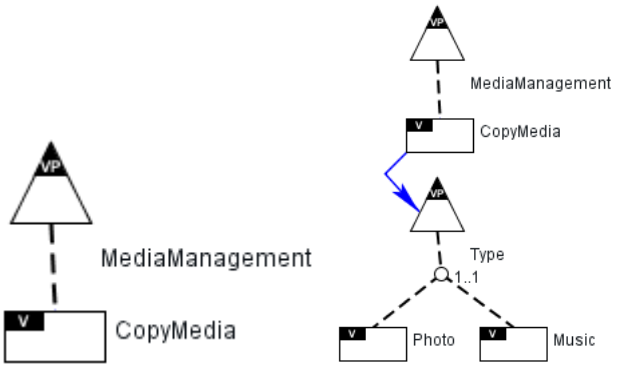

Figure 5.5: Copy media variant of the Mobile Media feature model

models and the efficiency of our FIG Basis Path method. To better analyze this correlation, we examine two different models of the CopyMedia variant of Mobile Media SPL application. Figure 5.5 shows both models of the copy media variant. Copy Media is an optional functionality that copies a specific media from one album to another. The model on the left side of the figure shows the original model. The right side of the figure shows the refactored model of the copy media variant. Since Copy Media is an optional feature of the Mobile Media SPL, the FIG Basis Path method may select the copy media feature only once for the original model which implies that this feature may be tested for one type of media only. Otherwise, for the refactored model, the FIG Basis Path method will select both variants (Photo and Music), and the feature will be tested for both types of media consequently. We conclude for RQ3 that the granularity of the feature model can impact the detection of our technique regarding fault efficiency.

5.7 Limitations

One limitation in our study involves the size of instances created. Since our methodology tries to maximize the number of features that are considered together in order

to reduce the total number of instances to cover all features, the first instance may be somewhat large and consequently the complexity of testing will increase. Thus, larger objects of study may lead to different results. Another limitation involves faults that might occur due to integration between features. Failures that can be found only during integration tests may not be found since we are discarding features that were previously tested.

Chapter 6

Conclusion and Future Work

In this thesis we have examined ways to improve the testability of software product lines. In this chapter we highlight our main contributions to software product line testing and present several areas to pursue as future work.

6.1 Conclusion

We have proposed a new definition for testability of software product lines based on the ability to re-use test cases between products without a loss of fault detection effectiveness. We built on this idea to identify elements of the feature model that contribute positively and/or negatively towards SPLs testability. We also developed key metrics to measure the impact of different approaches of designs of feature models for testing SPL and discuss the impact of the feature granularity of feature models on SPL testability.

Second, we provided a graph based testing approach called the FIG Basis Path method, which selects products and features for testing based on a feature dependency graph. This graph relies exclusively on the feature model of an SPL. However, the

FIG Basis Path method borrows ideas from white-box techniques, such as those that use control flow graphs and basis paths. This method should increase our ability to re-use results of test cases across successive products in the SPL family and reduce testing effort while retaining fault detection capability.

Finally, we report the results of a case study involving several non-trivial SPLs and show that for these objects, the FIG Basis Path method is as effective as testing all products in the SPL. 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 the products in our SPLs, and running only 10% of the test cases as on 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.

6.2 Future Work

This research on improving the testability of software product lines has highlighted many avenues and opportunities for future work. There are three main paths that we explain next.

First we will examine other variations of feature models such as $1...n$ relationships and the impact of constraints on the FIG Basis Path method. We plan to examine this technique on larger software product lines with more complex faults.

Second we plan to work on a fault model for SPLs. An initial fault model was proposed by McGregor [46]; however we believe that we can extend this model by analyzing objects fault types. With an accurate fault model, we will be able to

perform more thorough empirical studies in order to quantify the benefits of the FIG Basis Path method.

Finally, we plan to explore other aspects of testability, beyond those explored in this thesis. We will refine our metrics so that we can achieve better testability during the application engineering phase of SPL development by analyzing alternative feature models. We will perform a large-scale study of how feature granularity in feature models impacts testability in practice. Together these will lead to higher testability in software product lines.

Bibliography

[1] Steve Adolph, Alistair Cockburn, and Paul Bramble. *Patterns for Effective Use Cases*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[2] Randall C. Bachmeyer and Harry S. Delugach. A conceptual graph approach to feature modeling. In *Intl. Conference on Conceptual Structures*, pages 179–191, 2007.

[3] Don Batory. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[4] Don Batory. Feature models, grammars, and propositional formulas. In *SPLC*, pages 7–20. Springer, 2005.

[5] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[6] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, 2010.

[7] Antonia Bertolino, Alessandro Fantechi, Stefania Gnesi, and Giuseppe Lami. Product line use cases: Scenario-based specification and testing of requirements. In *Lecture Notes in Computer Science*, pages 425–445, 2006.

[8] Robert V. Binder. Design for testability in object-oriented systems. *Commun. ACM*, 37(9):87–101, 1994.

[9] Magiel Bruntink and Arie van Deursen. Predicting class testability using object-oriented metrics. In *SCAM '04: Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop*, pages 136–145, Washington, DC, USA, 2004. IEEE Computer Society.

[10] Isis Cabral, Myra B. Cohen, and Gregg Rothermel. Improving the testing and testability of software product lines. In *SPLC'10: Proceedings of the 14th International on Software Product Line Conference*, 2010.

[11] Samuel T. Chanson, Antonio Alfredo Ferreira Loureiro, and Son T. Vuong. On the design for testability of communication software. In *Proceedings of the IEEE International Test Conference on Designing, Testing, and Diagnostics - Join Them*, pages 190–199, Washington, DC, USA, 1993. IEEE Computer Society.

[12] Paul Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley, 2001.

[13] 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.

[14] David Cohen, Ieee Computer Society, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The aetg system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23:437–444, 1997.

[15] 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.

[16] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 129–139, New York, NY, USA, 2007. ACM.

[17] 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.

[18] Krzysztof Czarnecki. Overview of generative software development. In *Unconventional Programming Paradigms*, pages 313–328, 2004.

[19] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. In *Software Process: Improvement and Practice*, page 2005, 2005.

[20] Krzysztof Czarnecki, Steven She, and Andrzej Wasowski. Sample spaces and feature models: There and back again. In *Intl. Software Product Line Conference*, pages 22–31, 2008.

[21] Christian Denger and Ronny Kolb. Testing and inspecting reusable product line components: First empirical results. In *Intl. Symposium on Empirical Software Engineering*, pages 184–193, 2006.

[22] Institute O. Electrical and Electronics E. (ieee). *IEEE 90: IEEE Standard Glossary of Software Engineering Terminology*. IEEE Computer Society, 1990.

[23] Magnus Eriksson and Alvis Hgglunds Ab. Marrying features and use case for product line requirements modeling of embedded systems. In *Institute of Technology, UniTryck, Linköping University, Sweden*, pages 73–82, 2004.

[24] Magnus Eriksson, Jürgen Börstler, and Kjell Borg. The pluss approach - domain modeling with features, use cases and use case realizations. In *Lecture Notes in Computer Science*, pages 33–44. Springer-Verlag, 2005.

[25] Leire Etxeberria and Goiuria Sagardui. Quality assessment in software product lines. In *ICSR '08: Proceedings of the 10th international conference on Software Reuse*, pages 178–181, Berlin, Heidelberg, 2008. Springer-Verlag.

[26] A. Fantechi, S. Gnesi, G. Lami, and E. Nesti. A methodology for the derivation and verification of use cases for product lines. In *Software Product Lines*, Lecture Notes in Computer Science, pages 114–116. Springer, 2004.

[27] Eduardo Figueiredo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sergio Soares, Fabiano Ferrari, Safoora Khan, Fernando Castor Filho, and Francisco Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *Intl. Conference on Software Engineering*, pages 261–270, 2008.

[28] Roy S. Freedman. Testability of software components. *IEEE Trans. Softw. Eng.*, 17(6):553–564, 1991.

[29] Hassan Gomaa. Designing software product lines with uml 2.0: From use cases to pattern-based software architectures. In *SPLC '06: Proceedings of the 10th International on Software Product Line Conference*, page 218, Washington, DC, USA, 2006. IEEE Computer Society.

[30] Georg Grtter. Keynote outline split 2006 challenges for testing in software product lines, 2006.

[31] 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.

[32] Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc., New York, NY, USA, 1977.

[33] 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.

[34] Michel Jaring, Rene L. Krikhaar, and Jan Bosch. Modeling variability and testability interaction in software product line engineering. In *ICCBSS '08: Proceedings of the Seventh International Conference on Composition-Based Software Systems (ICCBSS 2008)*, pages 120–129, Washington, DC, USA, 2008. IEEE Computer Society.

[35] Meena Jha and Liam O'Brien. Identifying issues and concerns in software reuse in software product lines. In *ICSR '09: Proceedings of the 11th International Conference on Software Reuse*, pages 181–190, Berlin, Heidelberg, 2009. Springer-Verlag.

[36] 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.

[37] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 311–320, New York, NY, USA, 2008. ACM.

[38] Ronny Kolb and Dirk 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.

[39] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.

[40] Martin Griss Laboratory, Martin L. Griss, John Favaro, and Case Methodologist. Integrating feature modeling with the rseb. In *In Proceedings of the Fifth International Conference on Software Reuse*, pages 76–85, 1998.

[41] Kwanwoo Lee, Kyo Chul Kang, and Jaejoon Lee. Concepts and guidelines of feature modeling for product line software engineering. In *ICSR-7: Proceedings of the 7th International Conference on Software Reuse*, pages 62–77, London, UK, 2002. Springer-Verlag.

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

[43] Thomas J. McCabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, page 407, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[44] J. D. McGregor. Testing a software product line (cmu/sei-2001-tr-022). Technical report, Carnegie Mellon Software Engineering Institute, 2001.

[45] John McGregor. Testing a software product line. Technical report, Carnegie-Mellon University Software Engineering Institute, December 2001.

[46] John D. McGregor. Toward a fault model for software product lines. In *SPLC (2)*, pages 157–162, 2008.

[47] A. Metzger and P. Heymans. Comparing feature diagram examples found in the research literature. Technical report, Software Systems Engineering - University of Duisburg-Essen, February 2007.

[48] MicroEmulator. http://www.microemu.org/, 2010.

[49] Sonia Montagud and Silvia Abrah ao. Gathering current knowledge about quality evaluation in software product lines. In *SPLC '09: Proceedings of the 13th International Software Product Line Conference*, pages 91–100, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[50] H. Muccini and A. Van Der Hoek. Towards testing product line architectures. In *In: International Workshop on Testing and Analysis of Component Based Systems*, pages 111–121, 2003.

[51] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[52] Clmentine Nebut, Franck Fleurey, Yves Le Traon, and Jean-Marc Jzquel. A requirement-based approach to test product families. In *PFE*, volume 3014 of *Lecture Notes in Computer Science*, pages 198–210. Springer, 2003.

[53] Erika Mir Olimpiew and Hassan Gomaa. Reusable model-based testing. In *ICSR '09: Proceedings of the 11th International Conference on Software Reuse*, pages 76–85, Berlin, Heidelberg, 2009. Springer-Verlag.

[54] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31(6):676–686, 1988.

[55] Ovm improver. http://www.sse.uni-due.de/wms/en/?go=109, 2010.

[56] K. Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering*. Springer, Berlin, 2005.

[57] Klaus Pohl and Andreas Metzger. Software product line testing. *Commun. ACM*, 49(12):78–81, 2006.

[58] Klaus Pohl and Andreas Metzger. Variability management in software product line engineering. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 1049–1050, 2006.

[59] 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.

[60] Sacha Reis, Andreas Metzger, and Klaus Pohl. A reuse technique for performance testing of software product lines. In *Proceedings of SPLiT 2006 - Third Intl. Workshop on Software Product Line Testing*, pages 5–10, Baltimore, USA, 2006.

[61] Andreas Reuys, Erik Kamsties, Klaus Pohl, and Sacha Reis. Model-based system testing of software product families. In *CAiSE*, pages 519–534, 2005.

[62] Andreas Reuys, Erik Kamsties, Klaus Pohl, and Sacha Reis. Model-based system testing of software product families. In *CAiSE*, Lecture Notes in Computer Science, pages 519–534. Springer, 2005.

[63] Matthias Riebisch, Kai Bllert, Detlef Streitferdt, and Ilka Philippow. Extending feature diagrams with uml multiplicities, 2002.

[64] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature diagrams: A survey and a formal semantics. In *Intl. Requirements Engineering Conference*, pages 136–145, 2006.

[65] Andy Schürr, Sebastian Oster, and Florian Markert. Model-driven software product line testing: An integrated approach. In *Theory and Practice of Computer Science*, pages 112–131, 2010.

[66] Antti Tevanlinna, Juha Taina, and Raine Kauppinen. Product family testing: a survey. *SIGSOFT Softw. Eng. Notes*, 29(2):12–12, 2004.

[67] Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe composition of product lines. In *Intl. Conference on Generative Programming and Component Engineering*, pages 95–104, 2007.

[68] Unified modeling language. http://www.uml.org, 2010.

[69] Engin Uzuncaova, Daniel Garcia, Sarfraz Khurshid, and Don Batory. Testing software product lines using incremental test generation. In *Intl. Symposium on Software Reliability Engineering*, pages 249–258, 2008.

[70] Frank van der Linden. Software product families in europe: The esaps & caféé projects. *IEEE Softw.*, 19(4):41–49, 2002.

[71] J. Voas, J. Payne, R. Mills, and J. McManus. Software testability: an experiment in measuring simulation reusability. In *SSR '95: Proceedings of the 1995 Symposium on Software reusability*, pages 247–255, New York, NY, USA, 1995. ACM.

[72] Jeffrey M. Voas and Keith W. Miller. Software testability: The new verification. *IEEE Softw.*, 12(3):17–28, 1995.

[73] T. Wall. Abbot Java GUI test framework. http://abbot.sourceforge.net/doc/overview.shtml, 2010.

[74] Watson Wallace and Thomas J. McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric. *NIST SPECIAL PUBLICATION SP*, 1996.

[75] Jun Yan and Jian Zhang. An efficient method to generate feasible paths for basis path testing. *Information Processing Letters*, 107(3-4):87 – 92, 2008.