Winter 12-2011

Use of Constraint Solving for Testing Software Product Lines

Jiangfan Shi
University of Nebraska-Lincoln, jiangfan.shi@gmail.com

USE OF CONSTRAINT SOLVING FOR TESTING SOFTWARE PRODUCT LINES

by

Jiangfan Shi

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Doctor of Philosophy

Major: Computer Science

Under the Supervision of Professor Matthew B. Dwyer and Professor Myra B. Cohen

Lincoln, Nebraska

December, 2011

USE OF CONSTRAINT SOLVING FOR TESTING SOFTWARE PRODUCT LINES

Jiangfan Shi, Ph. D.

University of Nebraska, 2011

Advisers: Matthew B. Dwyer and Myra B. Cohen

A new software engineering methodology, software product line (SPL) engineering, has been increasingly studied in academia and adopted in industry in the past decade. It allows the delivery of similar, but customized, software products to customers in the same domain within a short time period. Software product line engineering produces an SPL by defining feature commonality and variability, and is supported by a well-managed asset base.

Based on case studies in the literature, in practice, SPL engineering can improve productivity from three to ten times, but in theory, it can dramatically push productivity to an extreme. The reasons for this high productivity root from both the variability and the ability to automatically generate configurations. Variability supplies a large configuration space, and automatic generation makes the configuration procedure trivial. High productivity, however, requires more efficient testing methods, so that we can ensure the correctness of SPLs with the same resource allocation percentage as in the traditional software engineering; traditional methods applied to SPL testing require a longer percentage of the software lifecycle.

In this dissertation, we show how modern constraint solvers can be used to tackle the challenge of efficiently ensuring dependability in SPLs from two perspectives: sampling and reuse. In sampling, the key is to choose a subset of products that are representative of the whole configuration space. We focus on one sampling technique, combinatorial interaction testing, that samples combinations of variability in the SPL. In reuse the goal is to leverage the inherent property of SPLs: similarity, which stems from the fact that

all configurations are generated from a core set of common and variable features. Our primary contributions are improved sample generation techniques for SPL testing that efficiently incorporate constraints between features, and reuse techniques that efficiently leverage similarities during integration testing.

More specifically, we propose several enhanced sample generation techniques for combinatorial interaction testing that leverage satisfiability solvers. Based on our empirical studies, we conclude that our techniques are efficient, and can generate high-quality samples in much less time from existing techniques that do not consider constraints. We then propose a compositional symbolic execution technique to achieve reuse during integration testing. A feasibility study shows that our technique is efficient, and can run as much as four times faster than a traditional directed symbolic execution technique.

ACKNOWLEDGMENTS

I would first like to sincerely thank my advisors, Dr. Matthew Dwyer and Dr. Myra Cohen, for their guidance, patience and support. I have been very fortunate to have them as my advisors as said in my very first email to them. They gave me much freedom to explore interesting problems of software testing areas, while providing invaluable insights and comments on my thoughts. My research has been benefited greatly from their breadth and depth of knowledge. Their high standards played a key role to build up my confidence in both the normal and research life, which is the most precious achievement I gained during the study. I also would like to thank my PhD committee members, Dr. Witawas Srisa-an and Dr. Jeonghan Ko, for giving me their valuable time and suggestions on my dissertation work.

I thank numerous developers and users of both Soot and JPF open sources. In particular, I would like to thank Richard Halpert for the technique discussions that I had with him through emails. My sincere thanks also go to the whole CSE department. They supplied a TA position to financially support my study, set up various interesting and fundamental courses to rich my knowledge, and provided technique support of computer facilities to speed up my experiments. I also thank Leping, Rahul and many other friends and families in the ESQuaRed lab, computer science and other departments. They made my life more colorful during the study. My sincere thanks also go to Deborah Ball Derrick for her professional English review and grammar improvement of my dissertation.

I would like to thank my whole family. During the hardest period of my PhD life, many my loved persons showed their support without reservations. My parents and my sister, Zaizhong, Fuzan and Jiangliu, suggested me to follow my heart and supported every decision I have made. My brother-in-law and parents-in-law, Qiwei, Changyi and Baolian, also showed their support by supplying detailed suggestions to handle

difficulties.

Finally, I thank my wife, Ting Wei, for everything she has done for me. She always stands behind me with strong support. Without her support, my life will be totally different. I also thank my lovely son, Benjamin (Zihan) Shi, who is one important source of my courage. His smile is the sweetest thing in the world.

This work was supported in part by NSF CCF through awards 0429149, 0444167, 0454203, 0541263, 0747009 and 0915526, by NSF through awards CNS-0720654, the U.S. Army Research Office through award DAAD190110564 and DURIP award W91NF-04-1-0104, the Air Force Office of Scientific Research through awards FA9550-09-1-0129, FA9550-09-1-0687 and FA9550-10-1-0406, the National Aeronautics and Space Administration under grant number NNX08AV20A, and by an NSF EPSCoR FIRST award. Any opinions, findings, conclusions, or recommendations expressed in this dissertation are my own, and do not necessarily reflect the position or policy of these organizations including NSF, ARO, AFOSR and NASA.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Ideally a software company should satisfy all requirements from all consumers in a market segment quickly. Some requirements are common, and some are variant. Common requirements lead to similarity among products, and variant requirements contribute to the uniqueness of each product. To maximize the reuse among these products, Software Product Line Engineering (SPLE) was invented to explicitly identify, manage and realize common and variant features throughout the software life cycle during requirements analysis, architecture design, implementation, testing and maintenance.

Many companies have adopted this methodology for developing their products. For example, based on case studies in [20], [19] and success stories in [109], Nokia can produce 30 mobile models which are three to six times the original number per year. HP can deliver a series of similar printers at a rate of two to seven times faster than before. Raytheon produces a product line of satellite ground control systems for the U.S. National Reconnaissance Office with a seven-fold productivity improvement.

With growing popularity in industry, we need to consider the unique properties of a Software Product Line (SPL) so that we can adapt current testing techniques or build new testing strategies to guarantee the correctness and high quality of an SPL. Next we show

the challenges faced during testing an SPL with a running example.

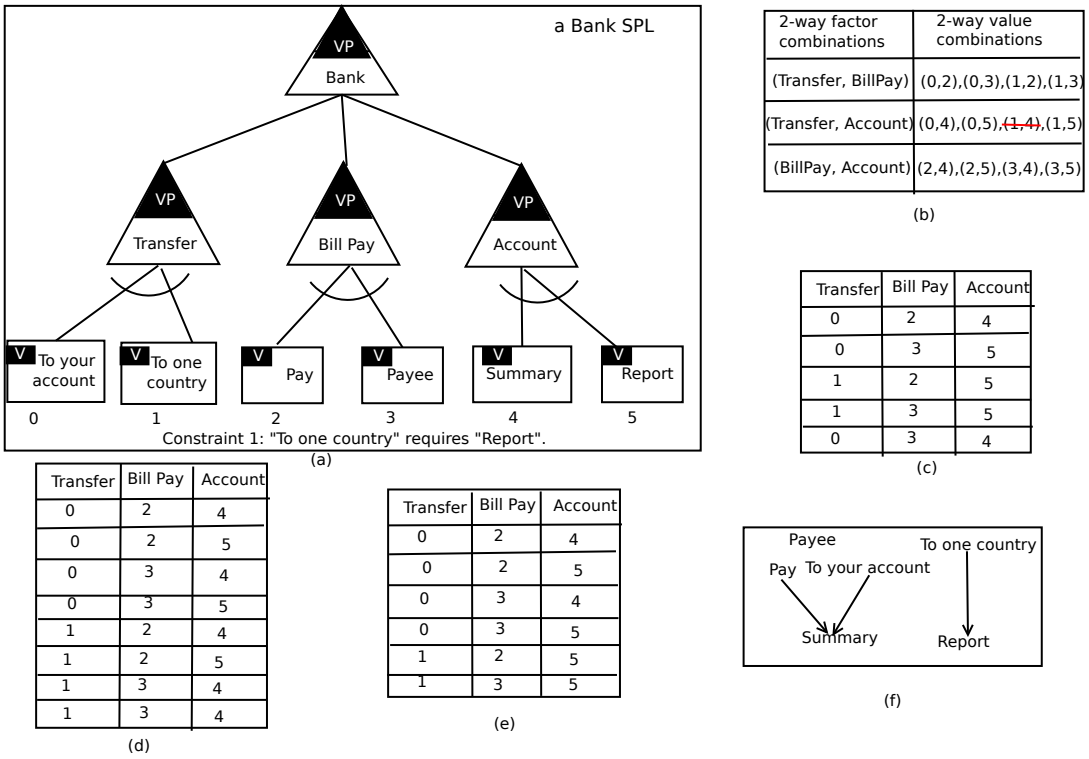
1.1 Motivating Example

In a traditional software development life cycle, testing is an important activity to find bugs before a product is released to the market. Testing occupies a lot of resources. For example, in 1999 Peters et al. [108] stated that 30 to 50% of development budgets were spent directly on software testing. In 2008, Zeller [154] found that validation (including debugging) could easily take up to 75% of development time.

In the past decade, many companies that have adopted SPL development have experienced positive results in terms of reduced time to market, as well as higher product quality. For example, Clements and Northrop [19] found an order of improvement between two and ten times compared to non-SPL development. In 2006, Hetrick et al. [62] described an incremental transition process to an SPL which removes redundant maintenance efforts over a common code base.

Under such reduced product generation time, if we still use the traditional method to test an SPL by testing each product one by one, then it is possible that the testing phase will become a bottleneck to releasing products to market. Consider the following example to illustrate this scenario.

Figure 1.1 (a) shows a synthetic small example of a bank SPL. There are six features organized hierarchically in a feature model. We describe feature models in Section 2.1.1 in more detail. Here we point out the meaning of the graphical notation. The solid line means a feature must appear in every product. In the bank SPL, *Transfer* must appear in all products, which in turn means either "To your account" or "To one country" must be chosen. The arc means that the features within its scope appear exclusively. Transfer is called a variation point which has two variants, "To your account" and "To one country."



2-way factor combinations	2-way value combinations
(Transfer, BillPay)	(0,2),(0,3),(1,2),(1,3)
(Transfer, Account)	(0,4),(0,5),~~(1,4)~~,(1,5)
(BillPay, Account)	(2,4),(2,5),(3,4),(3,5)

Transfer	Bill Pay	Account
0	2	4
0	3	5
1	2	5
1	3	5
0	3	4

Transfer	Bill Pay	Account
0	2	4
0	2	5
0	3	4
0	3	5
1	2	4
1	2	5
1	3	4
1	3	4

Transfer	Bill Pay	Account
0	2	4
0	2	5
0	3	4
0	3	5
1	2	5
1	3	5

Figure 1.1: Motivation Example

We can create eight products as shown in Table (d) in Figure 1.1 without considering the constraint listed at the bottom of the figure.

If we assume it takes ten days to release a product in a traditional software engineering methodology for one of these eight products, and we use 50% as the testing resource allocation rate, then we deduce that five days are used for development and another five days for testing. Based on an eight-fold magnitude productivity of SPLE, we further assume, after we switch to an SPLE for developing products, that we can produce these eight products in five days. If we do not change the testing strategy, then we need 40 days for testing. The overall testing resource allocation rate now is 88.9% (40/45) compared to the original 50% rate (5/10).

From this example, we can see that in the SPLE methodology, the testing phase can

become a bottleneck for releasing products to market quickly. The ability to be able to shorten the testing time is a challenging research problem in the SPLE community. The primary reason for such a lengthy testing time is the increased number of products. In an SPL with 30 optional features we may have as many as 1,073,741,824 valid products. In fact, the core challenges for testing an SPL are the exponential number of products and current testing techniques that focus only on the product level.

Our proposed solutions tackle these challenges from two perspectives: sampling and reuse. For example, Figures 1.1 (c) and (e) are two samples with respect to all products in (d) of the bank SPL. Sample (c) covers all 2-way feature interactions which are shown in (b). A k-way feature interaction refers to a combination of k features which may affect each other with their outputs. When a sample covers a k-way feature interaction, the k-way feature interaction embeds in at least one row of the sample. For example, there are four 2-way feature interactions between Transfer and Bill-Pay variation points as shown in 1.1 (b). Sample (c) covers the 2-way interaction (0,2) because this pair appears in the first row of the sample. Note there are only three 2-way feature interactions between Transfer and Account because (1,4) conflicts with the constraint. We can see that sample (c) has a size of five, which further reduces the testing resource allocation rate from 88.9% to 83.3% (25/30). Sample (e) covers the constraint which removes two products, (1,2,4) and (1,3,4). We get a sample size of six, which reduces the rate from 88.9% to 85.7% (30/35). With such a small model, it appears that we do not remove many products. However, in one subject of our experiment in Chapter 4, the GCC 4.1 optimizer, which has 199 variation points with 40 constraints, one constraint may remove 1.2×10^{61} number of products. The product space without considering the constraints consists of $2^{189} \times 3^{10}$ products, and testing this space is infeasible. With a sampling technique, we need only test 25 products to cover all 2-way feature interactions.

For reuse, we exploit inclusion relationships between small feature interactions and

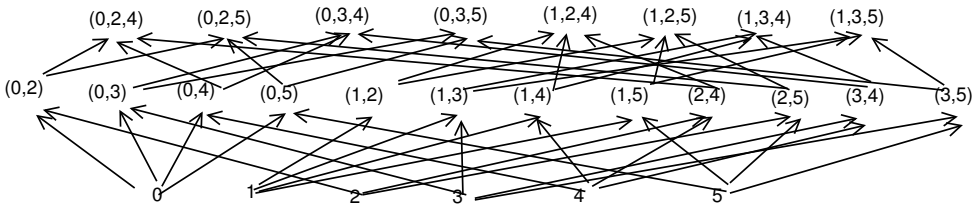


Figure 1.2: A hierarchical organization of feature interactions for the bank SPL

larger feature interactions. Figure 1.2 shows the inclusion relationships among single, 2-way and 3-way feature interactions for the bank SPL. We can see that feature 0 (To-your-account) is reused in four 2-way interactions, (0,2), (0,3), (0,4) and (0,5), and the 2-way interaction (0,2) is embedded into the two 3-way interactions, (0,2,4) and (0,2,5). With these inclusion relationships, we can reuse testing results from lower interactions to higher interactions. How to represent testing results and reuse them is a challenge. In this dissertation we introduce symbolic execution summaries and related composition technique to address this challenge. Next we present methods to test an SPL in this dissertation in detail.

1.2 Methodologies

Figure 1.3 provides an overview of the dissertation. There are three solutions, shown as Steps 1 to 3 in the figure. Coverage criteria is the first step to setup a testing scope, a subset of products, based on limited resources and predefined interesting testing objectives like all 2-way feature interactions. The sampling generation produces a subset of products to satisfy the criteria, and testing techniques can be applied to probe these products with the consideration of a reuse mechanism by exploiting similarities. Constraint solving is the core technique for supporting these three solutions, and the interaction perspective is our specific view for testing an SPL. We elaborate them in detail below.

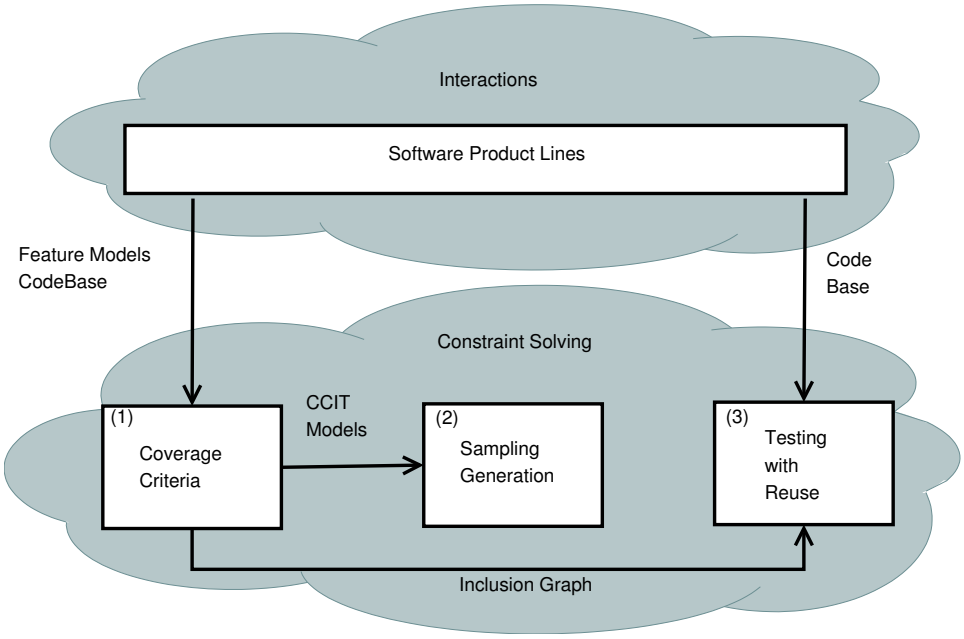


Figure 1.3: Methodology Overview

1.2.1 Establishing Coverage Criteria Related to Interactions and Constraints

Explicit variabilities lead to an exponential number of products and need special care from the testing perspective. In the requirement phase, there are many modeling languages to describe variabilities with a compact representation. For example, Orthogonal Variable Model (OVM) [109] is a graphic notation language that captures variability relations. With the same example, Figure 1.1 shows a feature model using the OVM notation for a bank SPL. We can see there are a total of six features. Each variation point, Transfer, Bill-Pay and Account, has two alternative features. All six alternative features consist of eight products without considering the constraints.

A variability model defines a scope of valid products for this SPL, which satisfies all implicit and explicit constraints. The above example includes only implicit constraints such as parent-child relations, alternative and mandatory relations. There could be other

constraints to filter more products. For example, Constraint 1 shown in the figure states that whenever the feature *To one country* appears in a product, then another feature, *Report*, must also appear in the same product. Hence after a client transfers money to another foreign country, there must be a specific report to the client. Considering this constraint, the number of valid products is reduced to six.

Here we propose a new testing coverage to exploit the variability model to cover feature interactions, constraints or both incrementally. We believe these feature interactions are essential sources for complicated bugs in an SPL. Organizing interactions systematically as a series coverage criteria is one of our contributions. We consider constraints as independent coverage criteria, and also setup mixed criteria with interactions. For example, we want to generate all interactions with two features, and there are 11 such pairs shown in Table (b) in Figure 1.1. Note that one 2-way feature interaction, (1,4), is removed due to a conflict with the constraint. Generating samples to cover feature interactions considering constraints is a new technique we have developed and discuss in Section 1.2.2.

We also propose directed feature interaction coverage by considering data flow directions among features. A k-way directed feature interaction extends a k-way feature interaction by considering how these k features affect each other in terms of data flow directions among them. For example, in Figure 1.1 (f), all directed relations between any two features for the bank SPL are explicitly shown as a feature dependence graph (FDG). We can see that an interaction between Pay and Summary is a single direction, so when we test this interaction we really need to trigger only a data flow from Pay to Summary. For an interaction between Payee and Summary, there is no real data flow between them, which means we do not need to test such an interaction at all. Note that the Payee feature is a unique feature in Bill-Pay for setting up items so that Pay knows the destination to pay money. Test cases do not need to be developed to probe this type

of interaction. We term the previous interaction as a feasible interaction, and later as an infeasible interaction. During testing, we only consider feasible interactions. Generating all feasible directional feature interactions and then testing them with a reuse mechanism is a new technique we have developed and presented in Chapter 5.

1.2.2 Generating Samples to Meet Coverage Criteria

After establishing this coverage, we further extend two existing combinatorial interaction testing (CIT) techniques, AETG-like and Simulated Annealing, to construct a sample to satisfy the interaction coverage criteria. The core technique contribution [27, 26, 28] is to enhance CIT as constrained CIT (CCIT) by integrating boolean SAT solvers to handle constraints appearing in feature models. For example, Figure 1.1 (c) shows one such sample which covers all valid 2-way interaction pairs shown in Figure 1.1 (b) with respect to the constraint.

In summary, we propose a new interaction-strength and constraint-sensitive coverage criteria for an SPL based on its variability model, and directional interaction coverage that considers data flows among feature interactions. We also introduce several related CCIT variants to generate samples to fulfill interaction coverage with optimizations. Next we introduce a new technique to fulfill directional interaction coverage.

1.2.3 Testing Interaction Trees by Exploiting Similarity

Before we introduce our technique to test directional interactions with a reuse mechanism, we want to mention another representation of inclusion relations other than hierarchical representation among interactions, i.e., partial products.

With relation to the properties of an SPL, there are many *similarities* among products, although each product is unique. Obviously, common features are the first source of

similarities. Some partial products contain variable features and can not appear in all products, but these partial products appear in many products. These partial products are a second source of similarities. The bank SPL in Figure 1.1 does not have any common features, but its eight products are similar due to shared partial products. For example, in Table (d) product (1,3,4) shares (1,3) with (1,3,5) and shares (1,4) with (1,2,4).

Both partial products and a hierarchical graph are good candidates for representing inclusion relations among non-directional interactions. For directional interactions, we need to expand this to add data flow relations. For example, Figure 1.4 is an extension of Figure 1.1 based on feature dependence relations in Figure 1.1(f). Figure 1.4 shows that there are three directional 2-way interactions, $\{pay \rightarrow Summary\}$ (PS), $\{ToYourAccount \rightarrow Summary\}$ (TS) and $\{ToOneCcountry \rightarrow Report\}$ (TR). There is also one 3-way interaction, $\{Pay \rightarrow Summary, ToYourAccount \rightarrow Summary\}$ (PS-TS). In this example, obviously there are two inclusion relations between two 2-way directional interactions, (PS,TS), to one 3-way directional interaction, (PS-TS), and we want to use testing results of both PS and TS to test PS-TS.

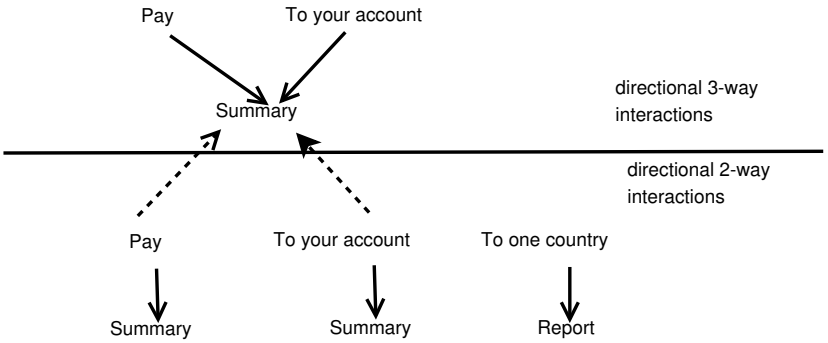


Figure 1.4: A hierarchical organization among *directional* feature interactions for the bank SPL

In Chapter 5 we discuss the detailed technical solutions.

1.2.4 Constraint-Solving-Centered Problems

All of these three solutions, interactions and constraints-oriented coverage definitions, CCIT sampling for constraints and inclusion-guided integration testing, are related to the constraint solving.

For the first solution, there are constraints in OVMs which are direct evidence of the existence of constraint solvers. In general, there are prevalent constraints in variability models. Tools such as FAMA [8] are needed to handle constraints so that consistency checking, specialization process and other reasoning tasks can be supported. In the dissertation, we need to consider constraint syntax transformation from *requires* and *excludes* in OVMs to the conjunctive normal form (CNF) for two Boolean SAT solvers, zChaff [95] and MiniSat [40], respectively. For the second solution, we integrate these two constraint solvers into an AETG-like algorithm for different optimizations. There may be millions of satisfiability checks during sample constructions. For the last solution, we employ symbolic execution to compute summaries for single features, and during the path exploration there are full of constraints collection and solving. We further compose summaries together for 2-, 3-, . . . , n-way directed interactions, which uses constraints concatenations, normalizations and solving often. Constraint solvers, Satisfiability Modulo Theories (SMT) solvers more precisely, could be Choco [15], CVC3 [33], Z3 [152] and other theory-oriented solvers.

Next we show our thesis, list four contributions and present the organization of chapters in this dissertation.

1.3 Thesis

The dissertation makes three theses shown below:

1. Our interaction-strength and constraint-sensitive coverage criteria provide a set of feasible targets to drive testing efforts to the core property of SPLs: variability. Our directional interaction coverage criteria further quantify interactions of SPLs explicitly, which can assist testers to focus on specific interaction patterns directly.

2. Our sampling techniques can test SPLs from the perspective of CCIT models more effectively than traditional techniques. The reason is because the techniques are integrated with SAT solvers tightly from three standpoints: 1) using returned true/false values from SAT solvers to construct valid configurations; 2) exploiting *Must* and *May* information from SAT solvers to ignore the satisfiability checking and to speed the construction; and finally 3) exploiting a whole model of SAT solvers to replace the construction after a certain threshold point.

3. Our integration testing technique can test directed interactions in SPLs more effectively than traditional techniques. The reason is because the technique exploits the similarity among directed interactions by reusing symbolic summaries of smaller-size directed interactions for composing summaries of larger-size directed interactions from the bottom up.

1.4 Contributions

The contributions of this research are four-fold:

1. Designing Coverage Criteria

   We refine a series of coverage criteria formally related to interactions and constraints, and develop a coverage computation technique to automatically capture coverage criteria explicitly. Chapter 3 introduces the technique in detail.

2. Generating Samples to Meet Coverage Criteria with a Focus on Constraints

   We design variant sampling techniques to generate a subset of products to meet the interactions and constraints coverage criteria. Several algorithms and empirical studies are introduced in Chapter 4.

3. Refining Coverage Criteria

   We define a coverage criteria formally to target directed interactions, and develop a coverage computation technique to automatically compute coverage criteria explicitly. Chapter 5 introduces the details.

4. Integration Testing SPLs Through Exploitation of Similarity

   We design a compositional technique to test directed interactions with a bottom-up reuse mechanism. Chapter 5 presents algorithms and evaluations.

1.5 Outline of Dissertation

We organize the remainder of this disseration into several chapters. Chapter 2 gives the background of our techniques from the above four perspectives with a focus on constraint-related issues. Chapter 2 also presents and discusses the state-of-the-art for testing an SPL. Chapters 3 and 4 introduce the coverage and sampling work, and then the integration testing work is described in Chapter 5. Finally, Chapter 6 concludes the dissertation by summarizing our contributions to both researchers and practitioners and by proposing future work.

Chapter 2

Background and Related Work

In this chapter, we first present a general introduciton of software product lines with a focus on feature models. Then we introduce the background used in our proposed techniques such as coverage criteria, sampling and integration testing techniques. Finally, we discuss most related work of testing SPLs in detail, and summarize the major differences between their and our techniques.

2.1 Software Product Lines

The objective of Software Product Line Engineering (SPLE) is to maximize reuse among a predictable set of similar products on the market, and the key is to manage and realize variability. Research in SPLE started more than decade ago, and related conferences have an increasingly impact on industry. For example, based on the history of the Software Product Line Conference (SPLC) website [125], SPLE started in 2000 in the U.S. and in 1996 in Europe under another name: software product family engineering. They were merged in 2005 to promote more communication among researchers leading to greater world-wide impact.

Based on the 2010 SPLC, many companies have adopted SPLE, including vehicle manufacturers such as Scania and Volvo [56], which focus on maintaining an evolved SPL architecture; NASA GSFC [47], which focuses on the testing part of an SPL architecture; a power and automation company, ABB [129], which focuses on realizing usability in SPL architectures; and a global positioning system (GPS) company, TomTom [123], which focuses on managed interfaces.

Many researchers have adapted different software-engineering procedures and techniques to manage and realize variability. For example, Jansen et al. [70] map features from the requirements level to the design and implementation levels. Pohl et al. [109] also illustrate realization of variability in different phases. Hendrickson et al. [61] provide semantic definitions in the architecture phase from the perspective of change sets and relationships. In addition to these traceability techniques, feature models – an important variability management tool, have been studied extensively. Next we give a detailed explanation of such languages.

2.1.1 Feature Models

Feature modeling is a key activity in an SPLE to manage variability, to define the scope in terms of the number of valid products for an SPL, to manage the complexity of a potentially huge number of features and relations among them, and to aid other activities such as requirements [72, 109, 89], design and architecture [110, 114, 130], and implementation [5, 76] to explicitly realize commonalities and variabilities.

Different notations and extensions have been proposed to capture various conceptual understandings, to enrich their expressiveness, and to meet application requirements among SPLs. The first feature model originated from feature-oriented domain analysis (FODA), and was first initiated in 1990 by Kang et al. [75]. Since then, there have been

many variants [34, 55, 142, 116, 36]. Kang et al. [75] introduce a tree structure, a feature diagram, to organize features with parent-children relations and three core expressiveness elements which still hold true in these extensions shown below:

1. the mandatory and optional relation for a single feature,
2. alternative and *or* relations among a group of features, and
3. other complicated constraints among features, including *require* and *excludes* relations.

Figure 1.1 (a) is an example for illustrating the above elements. For example, *Transfer* is a mandatory feature represented as a solid line. *To your account* and *To one country* consist of a group of features with the alternative relation, which is represented as an arc. Finally, the *Constraint 1* in the example shows an *requires* relation between *To one country* and *Report*.

Other researchers refine one of these three elements more formal or more detailed. For example, Czarnecki et al. [36] integrates a concrete constraint language, Object Constraint Language, from an UML language into feature models. This instantiates the third element specifically for expressing constraints. Czarnecki et al. [34] introduce cardinality-based expressions and split the alternative relations, the second element, to several more specific relations including exclusive-or, inclusive-or and exclusive-or group with optional sub-features. They also introduce other notations to make the feature model notation scalable, like referencing with recursions, cloning and labeling with attributes.

Besides efforts to build a richer set of feature modeling notations, researchers also focus on other related problems including reasoning mechanisms [36, 135, 8, 131, 81, 6, 144], formal semantic definitions [35, 6], a specialization process [34] and reverse engineering a feature model from the feature dependencies extracted from the code base [121].

For example, an impact analysis of changes of feature models, which may lead to a smaller or larger size of valid products, should feed results back to the engineers. Consistency checking is a basic analysis for validating a feature model, and has at least one configuration which satisfies all constraints.

All of these mechanisms use constraint solvers to tackle complicated relations in feature models; therefore, corresponding tools are integrated with different constraint solvers. For example, the feature modeling plugin for Eclipse in [36] uses a solver based on Binary Decision Diagram (BDD) [57] called Congit Software [31]. FAMA in [8] integrates three solvers, JavaBDD [73], JaCoP [69] and Sat4J [117]. The tool in [6] uses a logic truth maintenance system (LTMS) [45] for predicting which features should be chosen for a given set of chosen features based on the reasoning algorithm, Boolean constraint propagation (BCP). It also uses an SAT solver, Sat4J, for debugging a feature model by checking if it has at least one valid product. BCP is a core reasoning basis for Boolean proposition logic and a core algorithm for making modern SAT solvers fast. We give more detailed discussion in Section 2.3. The tool in [112] uses a prolog-based constraint solver.

We also present another model, the orthogonal variability model (OVM) [109], which expresses and describes the variability of an SPL using a different language. OVM supplies variation points to organize variants hierarchically, variants to represent a partitioned category, and constraints among them. A modeling tool, VarMod [143], is also available as an Eclipse plugin. Note that in our paper [25], we supply a formal explanation of OVM notations via a relational model, and more details are explained in Chapter 3. Although OVM does not originate from FODA and is different than feature models, there are mapping relationships with feature models among its core notations. For example, a variation point corresponds to non-leaf nodes which can have a group of sub-features; a variant may correspond to a non-leaf feature and leaf feature; and

constraints include *requires* and *excludes* which are the same as those in feature models. If a variant is a non-leaf feature, then it uses *requires* constraints to relate to another group of features. A more thorough comparison of feature models can be found in a survey paper by Schobbens et al. [118] in 2006.

For implementing a feature model at the code level, Thaker et al. [134] in 2007 tackled the safe composition problem related to a specific SPL development environment, Algebraic Hierarchical Equations for Application Design (AHEAD) [4]. AHEAD supplies an extended Java language, Jak, for supporting composition-based software development. In AHEAD each feature is implemented in a separate package and may depend on other features. When we compose different features together, there may be some compilation problems. For example, when we compose a feature A with another feature B to be a product, but this product does not include other parent features, which leads to the problem of references to undefined elements. They supplied a static analysis for collecting all dependent constraints related to feature implementation using Jak, and for proving with a SAT solver that a feature model can not generate a product which violates any of these rules.

2.2 Covering Arrays

In this section, we present definitions of covering arrays, applications of these covering arrays in the testing domain, and variant methods to generate covering arrays.

Informally, a covering array (CA) is an array with a size as $N \times k$, where N is the number of rows and k is the number of columns, which is called a factor in the CA literature. Each factor has the same set of values or choices. CAs can be used to cover all t-way value combinations (t-sets) of all t-way factor combinations, where t is called the strength. When t is 2, the CA is called a 2-way CA, which is most commonly used in

other applications and constructed. We give a formal definition below:

Definition 2.2.1 *A* covering array, $CA(N; t, k, |v|)$, *is an* $N \times k$ *array from a set,* v*, of symbols with the property that every* $N \times t$ *sub-array contains all ordered subsets of size* t *from the* $|v|$ *symbols at least once.*

For example, Figure 1.1 (c) shows a 2-way covering array where we have an input model as (N=5,t=2,k=3,v=2). There are eight combinations in total. Figure 1.1 (b) shows all 2-way factor combinations and then all corresponding all 2-way value combinations. Then (c) shows a covering array, which covers (b). For example, we can see that for (Transfer,BillPay), all four 2-way pairs (0,2), (0,3),(1,2) and (1,3) appear in the array.

The presence of constraints demands a new definition for a proper covering array. Integral to this definition is the concept of whether a t-set is consistent with a set of constraints.

Definition 2.2.2 *Given a set of constraints* C*, a given* t*-set,* s*, is* C*-consistent if* s *is not forbidden by any combination of constraints in* C*.*

This definition permits flexibility in defining the nature of constraints and how they combine to forbid combinations. We provide a definition of constrained covering arrays (CCIT), that is parameterized by C and its associated definition of consistency.

Definition 2.2.3 *A* constrained-covering array, *denoted* $CCA(N; t, k, v, C)$*, is an* $N \times k$ *array on* v *symbols with constraints* C*, such that every* $N \times t$ *sub-array contains all ordered* C*-consistent subsets of size* t *from the* v *symbols* at least *once.*

Besides these basic- and constrained-covering arrays, in some models, the strength t is higher for a set of factors and lower for other factors for different applications. These are not our focus in this dissertation, and more detailed definitions and examples can be seen in [27].

2.2.1 Applications of Combinatorial Interaction Testing

There are many applications for Combinatorial Interaction Testing (CIT). For example, in 1985, Mandl [88] applied the orthogonal Latin square, one stricter type of covering array, to design test cases for the Ada compiler. In 1992, Brownlie et al. [10] developed the orthogonal array testing system(OATS) to test PMX/StarMAIL at AT&T. In 1997, Cohen et al. [22] also applied the AETG algorithm to both a telephone switch software system and an Asynchronous Transfer Mode (ATM) network monitoring system.

Recently in 2004, Kuhn et al. [83] discussed *pseudoexhaustive* testing. They discussed previously published related work, which also analyzed a set of software systems. These systems include software systems on medical devices, remote agent experiment software on NASA' Deep Space 1 mission, and a set of POSIX operating system functions used by 15 commercial software systems. They then analyzed 329 error documents of a large distributed software system developed by NASA Goddard Space Flight Center. From these two analyses, they recommended that a covering array with a small t-strength like ($4 \leq t \leq 6$) is enough to exhaustively test software systems in practice. Of course, authors also mentioned that more experiments over other classes of software systems are needed.

In 2006, Yilmaz et al. [150] applied CIT to generate a sample with low strength, ran a configuration in this sample to collect bugs, associated bugs with a small portion of options, and finally supplied such options to developers so that they can use such fault characterization to find the bug location quickly. Originally they used all configurations to run, which is time-consuming and not scalable; with the sampling technique, they achieved almost as good results as all-configurations in terms of fault characterization. Another important finding is that CIT performed more consistently than random sampling for characterizing faults.

For sampling in the SPL domain, McGregor [91] applied a covering array to test

variability in an SPL in 2008. The variability is propagated from requirements through architecture to implementation. For architecture, there maybe variation points which have several components to implement the same function with other requirements. For example, there are different communication protocols like GPRS, EGPRS and UMTS in the wireless device domain. Sampling techniques can be used to generate a subset of products with a t-strength coverage from a practical standpoint.

In 2008, Xiao et al. [113] applied CIT to test configurable software systems by considering different prioritization techniques of regression testing. In 2010, Si et al. [67] applied CIT for GUI testing, and then devised a genetic algorithm for repairing generated covering arrays. In 2011, Yuan et al. [151] developed a new coverage criteria considering specific properties of GUI testing such as event combinations and event sequence length, and then applied CIT to generate these covering arrays.

Next we focus on the introduction of different methodologies for generating a covering array from four perspectives: algebraic methods, meta-heuristic search, constraint solving and greedy methods.

2.2.2 Algebraic Methods

In 1999, Stevens et al. [128] discussed constructive methods to build covering arrays. Usually constructions find a smaller size of covering array, but this can not be generalized. Practioners need to understand when a Combinatorial Interaction Testing (CIT) problem can use a constructive method or not, which requires deep understanding of the method. In 2004 and 2005, Hartman et al. [60, 59] also discussed constructive methods. Because these methods are not the focus of our algorithms, here we do not discuss more related papers in the literature.

2.2.3 Meta-Heuristic Search Methods

In 2003, Nurmela [102] introduced a Tabu search method to find an new upper bounds for some previously constructed CIT problems. Tabu search [50] is a meta-heuristic search to target an optimization problem as an improved version for a local search. One unique property is that a tabu list, including recently visited solutions, can be used so that local optima can be avoided and a global optimal solution can be found.

In 2003, Cohen et al. [23, 24] introduced simulated annealing (SA) [98] and genetic algorithms (GA) [43] to construct covering arrays. SA avoids the local optimum problem with a probability which depends on a global temperature variable, T. At the beginning T is big and leads to a large freedom to move from a good solution to a bad solution. As T decreases, the possibility is smaller which gradually prevents such a bad move. After a fixed number of iterations, an optimal or good solution is found from an initial solution. GA is another meta-heuristic algorithm to mimic a natural evolution process. GA has several unique operations to find a final optimal solution, including inheritance, mutation, crossover and selection. The basic principle is that the next generation is better than previous predecessors. The inherit operation keeps good merits from predecessors; the mutation operation changes some parts of a gene in the hope of finding better solutions. The crossover operation exchanges parts between two genes to put both good parts together for a better gene, and the selection operation selects good candidates from a gene pool. Based on results, they suggested that heuristic search techniques can produce a smaller size covering array than greedy techniques. We also confirmed this claim in our experiments [28, 27, 26]. In 2011, Garvin et al. [48] improved the efficience of SA, which was observed to produce smaller covering arrays with much worse speed in our paper [26].

2.2.4 Constraint Programming

In 2006, Hnich et al. [63] represented a covering array problem with a constraint programming problem, and used several constraint models to solve the construction problem more efficiently. This work is close to our extended greedy algorithms to handle constraints, and we give more detailed discussions in related work section for this work. In 2010, Oster et al. [103] used constraint programming to construct pair-wise covering arrays incrementally.

2.2.5 Greedy Methods

There are two primary classes of greedy algorithms that have been used to construct covering arrays. The majority of algorithms are the one-row-at-a-time variation of the automatic efficient test case generator (AETG)[22]. A different type of greedy algorithm is the In Parameter Order (IPO) algorithm [132] and In Parameter Order General (IPOG) [86]. Rather than focusing on a row-at-a-time, the IPO algorithm generates all t-sets for the first t factors and then incrementally expands the solution, both horizontally and vertically using heuristics until the sample is complete.

The deterministic density algorithm (DDA) [29] introduced another requirement, repeatability, for covering array techniques. As discussed by Tang et al [136] in 2000, this deterministic property can be used to help fix bugs by repeating problems. DDA extends AETG based on the observation that when we bind a value for a factor in AETG, the criteria is related to the number of uncovered t-tuples between this factor with each *previous* factor-value binding, and we do not consider the number of uncovered t-tuples for all other remaining factors that have not yet been bound with values. Not exploiting the global view among all uncovered t-tuples is one *limitation* of AETG, which is an inherent property of a greedy algorithm. Then Colbourn et al. [29] continued to propose

two concrete properties, local density and global density, to measure such global view. Furthermore they supplied an implementation for constructing covering arrays with an approximation of these two properties. From experiments, DDA seems to be not a strong winner among other techniques including AETG, TCG [136], IPO [132], TConfig [146]. This is reasonable because the problem of choosing a row that can maximally cover un-covered t-tuples is NP-Complete by itself, as proved in [29].

Fore more techniques about combinatorial testing, interested readers may read a survey [54] by Grindal et al. in 2005 or a more recent survey [101] by Nie and Leung in 2011.

In summary, much of this literature ignores practical aspects of applying CIT to real systems, which limits the effectiveness and applicability of this work. In this dissertation, we focus on one difficult, yet prevalent, issue which may confound existing algorithms – the handling of constraints. More detailed discussions are in Chapter 4.

2.3 Constraint Solving

The satisfiability (SAT) problem determines whether a Boolean formula can possibly be true, that is, if there is at least one assignment that satisfies all of the conditions. It was introduced in computation theory by Cook and Levin around 1973. In computation theory SAT is interesting because it was one of the first NP-Complete problems. Many other problems, such as the clique and subset-sum problems, are proven NP-Complete by reducing from SAT.

SAT can be applied to many problem domains, especially Artificial Intelligence and Electronic Design Automation(EDA). In fact those domains drive the evolution of SAT techniques. For example, in EDA combinatorial equivalence checking, microprocessor verification and field-programmable gate array routing problems [99] are typical SAT

applications. Recently SAT has also been applied to software test generation and software verification. For example, in [127, 84, 3] a test suite can be generated by a SAT solver directly. Clark et al. in [16] model checkers used a SAT solver for bounded model checking.

The basic algorithm for solving SAT problems is due to Davis and Putnam (DP) [39]. In order to overcome the memory limitation of the DP method, Davis, Logemann and Loveland (DLL) [38] proposed a tree-like searching algorithm, called DLL. DLL systematically searches for a truth assignment that makes the formula true. Most modern SAT solvers [155, 95, 40, 90, 41, 117, 74] inherit the propagation mechanism, Boolean Constraints Propagation(BCP), from the DLL algorithm. They also extend the DLL algorithm for better performance with other features, such as the 2-literal watching scheme, non-chronological backtrack, clause learning and restart. We call these algorithms DLL-like algorithms. With modern SAT solvers, problems with thousands of variables can be solved in seconds.

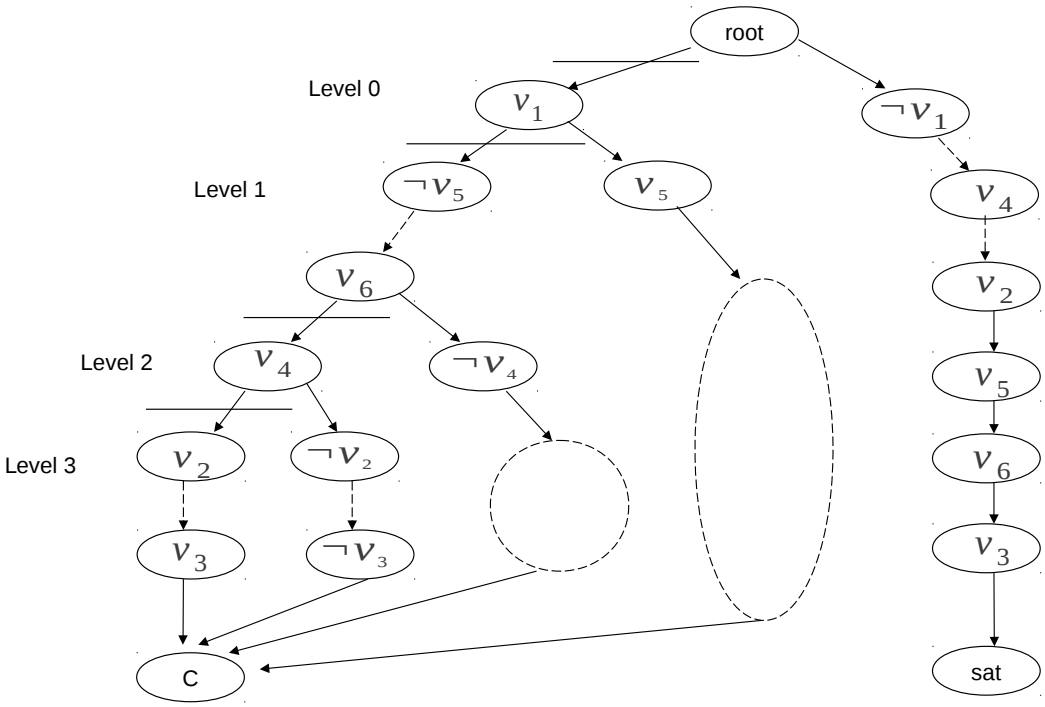
2.3.1 Davis-Logemann-Loveland Algorithm

The original DLL algorithm [38] forms the framework for almost all modern SAT solvers. DLL divides the search process into several steps, such as preprocessing, variable binding, BCP and backtrack. Preprocessing is a step to simplify the input formula before a SAT solver tries to search for a model. Variable binding is a step that chooses a truth-value for a free variable. BCP applies the unit rule to the clause database to get more variable bindings. The *unit rule* is the core reasoning basis. The unit rule involves two conditions: 1) all other literals are bound with false in a clause; and 2) the clause must be true. Under such a situation, the last remaining literal must be bound to true. Backtrack is a step to resolve a conflict encountered in the BCP step. Most modern SAT solvers extend the

feature in different steps, usually in the BCP and backtrack steps which are the two most computationally-expensive steps in DLL. Before we introduce these steps in detail by following the pseudo code in Algorithm 1 with the running example shown in Figure 2.1, we explain several important notations in Figure 2.1.

Figure 2.1 shows a search tree for the formula ϕ with a CNF format. Each path represents a variable binding assignment, which may end with SAT or UNSAT (Conflict). There are two important notations: levels and dashed/solid lines. A level consists of a group of variables, and levels are numbered starting from 0. When the first variable in a level is bound to true/false, then all following variables in the same level are bound to true/false automatically by BCP. For example, Level 1 includes two variables, v_5 and v_6. When v_5 is bound to false, based on the last clause in ϕ, $(v_6 \vee v_5)$, BCP deduces that v_6 must be bound to true. For the simplicity of the search tree, we only annotate levels for the left-most path. A dashed line from a bound variable to another bound variable refers to the implication relation between these two variable assignments. With the same example, there is a dashed line from $\neg v_5$ to v_6. A solid line from a bound variable to another bound variable represents an independent assignment for the latter variable during the tree exploration. All initial variables of levels accept solid lines. There are also two special nodes, C and *sat*. C represents a conflict for a given partial assignment, and we discuss the concept in this section soon. The *sat* simply indicates a satisfiable complete assignment for a formula.

For Algorithm 1, it first simplifies an input formula at statement 2. For example, it scans the formula to find two complementary clauses, c and $\neg c$. If there exist such clauses, the formula is trivially unsatisfiable because we cannot make both true under the same variable bindings in both clauses. If DLL finds v and $\neg v$ in the same clause, it just removes the clause from the clause database. The reason is that with any assignment of variable v, the clause is always true.



$$\phi = (v_1 \vee v_4)(v_2 \vee v_3)(v_2 \vee \neg v_3)(\neg v_1 \vee \neg v_2 \vee v_3)(\neg v_1 \vee \neg v_2 \vee \neg v_3)(v_5 \vee v_6)$$

$$vbs_l = v_1 \mid \neg v_5 v_6 \mid v_4 \mid v_2 v_3$$

$$vbs_r = \neg v_1 v_4 v_2 \mid v_5 \mid v_6 \mid v_3$$

Figure 2.1: The DLL Search Algorithm

```
DLL Algorithm(φ)
Require: Input: a CNF formula φ, Output: Satisfiable or Unsatisfiable
 1: preProcessing(φ)
 2: initialize(φ,vbs_s,vlist,g)
 3: while True do
 4:     if freeVariable(v) then
 5:         BCP(vlist,v,conflict,v')
 6:     else
 7:         return Satisfiable
 8:     if conflict then
 9:         Backtrack(v',backlevel)
10:         if backlevel<0 then
11:             return Unsatisfiable
```

Algorithm 1: DLL Algorithm

Then it initializes three global variables: a VBS variable vbs_s, a variable list *vlist* and a basic implication graph g. We use Variable Binding Snapshot(VBS) to express the progress

of a current search procedure for total binding. For example, in Figure 2.1 there are two *VBS*s in the graph, vbs_l and vbs_r. vbs_l represents a variable binding snapshot when the SAT solver reaches the end of the left-most path, and vbs_r the right-most path. vbs_l also indicates a total binding order. For example, v_1 is on the left of v_5 which means v_1 has been assigned a truth-value before v_5.

Initially there is no assignment inside the vbs_s. For the $vlist$ it includes all variables in ϕ. For each variable v there are two associated lists, the positive list and the negative list. The positive list includes all clauses with the literal v, and the negative list includes all clauses with the literal $\neg v$. For example, given a CNF formula ϕ in Figure 2.1, for the variable v_1, the positive list includes the clause $(v1 \vee v2)$ and the negative list includes two clauses $(\neg v1 \vee \neg v2 \vee v3)$ and $(\neg v1 \vee \neg v2 \vee \neg v3)$.

Third, it enters into the main part of the algorithm. It will stay in the *while* loop until it finds a model or it will search the whole assignment combination space without finding a model. When there is no free variable left, the SAT solver finds a model. In statement 4, the freeVariable() returns false under such situation. Then the flow goes into statement 7, which returns SAT. When it does not find a model, it will continue to bind a true/false value to a chosen free variable, v, in freeVariable(). There are different techniques to choose a variable in a formula, such as the Variable State Independent Decaying Sum (VSIDS) invented in zChaff [153], which employs the dynamic information of the search process.

After freeVariable() produces a new decision variable binding, the BCP() begins to propagate the binding information v according to the unit rule so that more variable bindings are implied. The BCP() function has three parameters: the new variable binding v, the conflict indicator *conflict* and the conflict variable v'. v is the input parameter. The *conflict* and v' are the output values returned from the BCP() function. *Conflict* indicates whether the BCP encounters a conflict during the propagation process. A *conflict* is a

situation where the same variable is implied to have both true and false values based on the unit rule. If so, the *conflict* has a true value; otherwise the *conflict* has a false value. If the *conflict* has a true value, then the v' records the conflict variable during this BCP propagation process. BCP is an iterative procedure, and it continues to propagate the generated implied variable binding until there is no variable bindings that can be implied or it reaches a conflict.

When there is a conflict, the algorithm tries to backtrack. This is done in statement 9 with the backtrack() function. Most backtrack algorithms in modern SAT solvers are similar. There are two backtrack methods: chronological backtrack and non-chronological backtrack. The latter method resolves the conflict much more effectively because it will drive the search to skip parts of the assignment, which lead to the same conflict again. For the chronological method, backtrack returns the parent level; for the non-chronological method, it may return several levels higher to correct a conflict with a binding for a variable.

If backtrack returns a level k, which is equal to or greater than 0, then backtrack undoes all the bindings greater than and equal to k and starts the search from level k. This can be seen from the statement flow from 10 to 4 where the backtrack level is greater than or equal to 0. If backtrack returns a level smaller than 0, then the formula is unsatisfiable because it fails to bind a truth-value to the decision variable in level 0. This can be seen in statement 11, when SAT solver stops. Next we explain BCP and Backtrack in more detail.

2.3.2 Boolean Constraint Propagation

BCP is a procedure that propagates the current variable bindings to acheive more forced variables bindings. Without the BCP procedure, some variable bindings that lead to unsatisfiable assignments will be taken. Obviously this will waste time. As discussed in

the DLL framework, BCP starts from a decision variable binding, and applies the unit rule to get the implied variable bindings. Decision binding and BCP are two sources driving the search. Another source is the backtrack mechanism, when there is a conflict during BCP procedures.

According to experiments in [156], much time is spent in the BCP procedure during the search. Sometimes even 80% of the execution time is spent in the BCP. Therefore, researchers use different engineering strategies to improve the efficiency of the BCP. More specifically, we need to facilitate BCP to quickly find the unit clauses.

2.3.3 Backtracking

Backtracking is a procedure that decides where the SAT solver should restart the search when the SAT solver encounters a conflict. There are two kinds of backtrack: chronological and non-chronological. Chronological backtrack is used in the DLL algorithm [38]. Non-chronological backtrack is an improvement of chronological backtrack. It was first mentioned in RelSat SAT solver [74], and is used in almost all modern SAT solvers.

Clause learning is a procedure to analyze a conflict. It returns a set of variable bindings as a reason for the conflict. Clause learning is the core of the non-chronological backtrack algorithm. Clause learning not only helps SAT solvers skip some variable bindings which lead to unsatisfiable, but also stops the same conflict from happening in future searches.

2.4 Symbolic Execution

Symbolic execution (SE) is a technique to explore all behaviors of a program in terms of a path tree. Because SE aims to execute all feasible paths at once, it sets up an ideal and exact upper bound for path coverage. The method that SE takes is to use symbolic values to represent all possible choices for parameters, to collect all path conditions whenever

there is an explicit or implicit branch point, and then to check if such path conditions are satisfiable via a constraint solver (a decision procedure). Because of the difficulties of such underlying constraint solving and the exponential number of paths, SE does not usually scale well to large size programs as mentioned in [14] by Cadar et al. in 2011. SE was first proposed by King [80] in 1976 for automatic program testing; there has not been much progress in academia until recently, which has been motivated by advanced constraint solving and better understanding of programs. Also, because of SE's interesting machinery and its inherent ultimate goal, many researchers are now focusing on these two challenges: an exponential number of paths and complicated constraints.

2.4.1 Tackling a Large Path Space

In 2007, Godefroid [52] tackled the problem of a large path space by using summaries of functions during the path exploration procedure to avoid recomputing the same functions when SE encounters them again. The summary for a function represents all paths in a function. As mentioned by Person et al. [106], it is probable that an incomplete summary for a function is computed due to loops, iterative calls, un-bound complex data structures like linked-list or simply run-out-of-time. One inherent property of this method is to push the exploration of an exponential number of paths from SE to an underlying decision procedure.

In 2008, Cadar et al. [13] tackled the problem by skipping paths, which had the same side-effects as one of the explored paths before. The key decision is to check if one incomplete path, p_1, has the same side-effects as one of the completed paths, p_2. If so, we can stop to continue the exploration of p_1 which may involve many paths. The method they use is to partition a method into several sections following the original sequential order, where each section corresponds to a partition of symbolic parameters. In order

to split a method into two sections, there should be no interactions between these two sections. More specifically, there should be no live variables in Section 1 with respect to Section 2. If such situation occurs, then we can separately transform one method to two smaller methods and symbolically execute them separately. With such technique, we can explore more distinguished paths than the original blind-search SE, which may lead to more branch coverage and expose more bugs as shown in their experiments.

In 2011, Person et al. [107] took advantage of the evolution of programs in terms of versions, which encompass a small set of changes between each versions. The impacted code area of these changes can be used to guide the symbolic execution over only these areas, and skip previously explored areas for which the impact analysis (static analysis) can guarantee no further errors introduced by these changes.

2.4.2 Tackling Complicated Constraints

In 2007, Shannon [120] proposed a String theory with a finite-automata representation to capture and reason operations over *strings*. The operations include methods *equal*, *charAt*, *contains* and *concat* of java.lang.String interface. In 2008, Kroening and Strichman [82] illustrated many different decision procedures including linear arithmetic, array, bit-vector, pointer and other mixed theories like string with linear arithmetic theory. In 2009, Belt et al. [7] provided a light decision procedure to check the satisfiability of a path condition faster. Their theory is a linear arithmetic theory.

2.4.3 Applications of Symbolic Execution

In 2006, Yang et al. [148] applied SE over the disk mounting code of three widely-used Linux file systems, ext2, ext3 and JFS, and found bugs in all of them. In 2008, Person et al. [106] applied SE over a pair of methods, such as a method with its refactored counterpart,

to check if they are semantically equal and if not, compute the difference in terms of paths. This technique is more precise than simple text differential tools like diff because it provides a path-by-path comparison between two methods. In 2008, Csallner et al. [32] applied SE to collect invariants of programs for a set of runs of a given test suite. SE has already explored many paths for a program, and each path is general in term of representing a lot of inputs. These inputs trigger the same path; we say that these inputs are in the same partition. When compared with Daikon [42], they can construct more *relative* invariants in terms of paths, or more specifically a set of pairs (precondition, post-condition).

There are many tools in both academia and industry, which include Symbolic PathFinder [100] from NASA, Cute and JCute from UIUC [139], Crest [138] and BitBlaze[137] from UC Berkeley, Klee [126] from Stanford, Pex [92] and Yogi [93] from Microsoft, and Appolo [2] from IBM.

In summary, there are many new techniques to address two basic challenges: a huge number of paths and complicated constraints. We just give a very small discussion in this domain, and there are many other applications and techniques of SE. You can find them in a comprehensive survey on SE by Pasareanu et al. [111] in 2009 and by Schwartz et al. [119] in 2010. Next we introduce definitions of symbolic method summary, which is used in Chapter 5.

2.4.4 Symbolic Method Summary

Several researchers [53, 106] have explored the use of method summarization in symbolic execution. In [53] summarization is used as a mechanism for optimizing the performance of symbolic execution whereas [106] explores the use of summarization as a means of abstracting program behavior to avoid symbolic execution. We adopt the definition of

method summary in [106], but we forgo their use of over-approximation.

The building block for a method summary is the representation of a single execution path through method, m, encoded as the pair (pc, w). This pair provides information about the externally visible state of the program that is relevant to an execution of m at the point where m returns to its caller. As described above, the pc encodes the path condition and w is the projection of s onto the set of memory locations that are written along the executed path. We can view w as a conjunction of equality constraints between names of memory locations and symbolic expressions or, equivalently, as a map from locations to expressions.

Definition 2.4.1 (Symbolic Summary [106]) *A symbolic summary, for a method m, is a set pairs $m_{sum} : P(PC \times S)$ where*

$$\forall (pc, w) \in m_{sum} : \forall (pc', w') \in m_{sum} - \{(pc, w)\} : pc \wedge pc' \textit{is unsatisfiable}.$$

Unfortunately, it is not always possible to calculate a summary that completely accounts for the behavior of all methods. For example, methods that iterate over input data structures that are unconstrained cannot be analyzed effectively – since the length of paths are not known. We address this in this dissertation using the standard technique of bounding the length of paths that are analyzed.

After presenting the background including feature models, covering array, constraint solving and symbolic execution for this dissertation, we discuss related work on testing an SPL next.

2.5 Testing Software Product Lines

Coverage criteria guide the testing effort of a software system to a specific quatified property with the hope of detecting more bugs in less time. These coverage criteria can be represented in different granularities, including statements, branches, paths for traditional software systems. In this section, we discuss proposed coverage criteria special for SPLs. All testing techniques attempt to reduce redundant testing efforts by reusing testing results. For SPLs, there are also techniques focusing on common features for saving testing efforts. We also discuss such related techniques in this section.

2.5.1 Testing SPLs from a Coverage Perspective

In 2010, Cabral et al. [12] presented a technique to reorganize a feature model as a feature inclusion graph (FIG) for explicitly showing the testability of an SPL. The testability is related to determining the minimum number of basis paths [147] for covering all features. If the number is small, then the testability of an SPL is good; otherwise, the testability is bad. This indicates that their algorithms construct the longest paths first to cover as many features as possible. Based on FIG, they proposed three new coverage criteria: FIG basis paths, FIG grouped basis paths and all-features. Based on their experiments, the FIG basis paths coverage is a good indicator to find all faults with a smaller number of products. Compared with the work in Chapter 5, this coverage does not consider choosing products for targeting interactions and constraints, which are semantic behaviors related to an SPL. Instead their coverages consider a static structure among features. From a practical perspective, they are complementary with ours. Such lighter coverage can be used in the early stage for several initial versions of an SPL. After the SPL seems to be stable in terms of the number of found bugs, we can then apply our more thorough coverage criteria.

Kauppinen et al. [77] proposed two coverage criteria: hook coverage and template

coverage. Template coverage is at the SPL level, and hook coverage is for a single product. The template refers to a method in a common code which invokes hook methods, which have different implementations for different products. Usually a hook method is inside an abstract class, which is then implemented with different concrete classes for different products. There are method and class coverage for both template and hook coverage. For the method coverage of a template, it is defined as the covered number of hook invocations in the template method divided by the total number of hook invocations. For the method coverage of hook, it is defined as a traditional structure coverage, like statement coverage. Compared with the dissertation work in Chapter 3, these coverage criteria are relative to a restricted implementation of an SPL, a framework-based development, whereas ours are not limited to such special, albeit widely used, development methodology. In fact, ours focuses on the requirements level, feature models, and theirs focuses on the implementation level. From the interaction perspective, ours covers all-way interactions systematically, and theirs does not have such an objective. However, if we associate their coverage criteria with interactions, then theirs targets only 2-way interactions between common features and variable features. They developed a tool, RITA [133], for supporting different engineering phases in SPLE. In the testing phases, RITA applies their testing coverage criteria.

Muccini and van der Hoek [97] provided a brief overview of many of the issues from an SPL architecture perspective. They provided suggestions for adapting traditional testing methods for handling SPL properties, variability and commonality, which correspond to common and optional components. For unit testing, there is no difference for common and variant components, but they suggested a higher priority for common components. For integration testing, they proposed to combine common components first and then integrate them with other optional components with a big bang strategy. They also mentioned a sampling method suggested by McGregor [91] with a small adaption

for integrating common components first. In conformance testing, they mentioned conformance between an implementation and at least one architecture, and conformance between an architecture and an overall product line architecture in terms of constraints. In regression testing, they classified three situations: 1) between two product architectures; 2) between two implementations for two architectures; and 3) between two programs for the same product architecture. Finally they proposed the challenge of testing a product line architecture in term of exponential number of architectures, and proposed a direction by exploring similarity among them.

Compared with the dissertation work in Chapter 3, they mentioned the large space of variability combinations as a key challenge, but did not propose test coverage approaches that allow for cost-effective trade-offs as we do. They described how testing might be targeted at portions of new product line instances that have yet to be considered, but they did not propose a framework for cumulative test coverage that will enable targeted testing.

The technical report of McGregor [91] proposed a systematic way to test an SPL. It is the most closely related to ours in that it considers the connection between combinatorial interaction methods, in his case orthogonal arrays, to cover the space of SPL variability points [91]. He did not, however, provide details on how SPL models such as OVM can be mapped onto appropriate representations to allow interaction methods to be applied, nor did he address the significant challenges arising from the use of constraints in those models. Finally, he did not develop the connection between interaction methods and test coverage criteria.

2.5.2 Testing SPLs from a Similarity Perspective

In the V-model there is a testing phase corresponding to use cases. Test cases are represented as a sequence of steps which correspond to normal English sentences. For similar use cases, these test cases are also similar in terms of steps. In 2004, given a set of similar test cases, Geppert et al. [49] extracted variabilities as a pair of parameters and values, decomposed these steps as common and variable steps where parameters are used, and then managed these parameters with a decision tree where each node corresponds to a parameter and each out-edge corresponds to a concrete chosen value for the parameter. Finally, they automated the test case generation by exploring this decision tree and collected concrete steps in which parameters are bound with values. This is a technique to improve a typical manual black-box testing to address variability and similarity. One limitation is that users have already written a set of usual test cases, and it is possible that the technique can generate the template and decision trees directly from use cases.

In 2004, Bertolino et al. [9] captured commonality and variability in the use case representation at a software requirement phase. They extended Cockborn's use case [21] template with tags such as alternative tags, parametric tags and optional tags, which capture variability of an SPL in the requirement level. Then they used category partition methods to develop a test specification, which setup an upper bound of all possible scenarios for a use case within a set of different products. From their methodology, we can see that common description of a use case is shared with a set of related products without rewriting. Compared with the work in Chapter 5, we also exploit commonality, but we focus on the integration testing over a code representation, where commonality and variability features are represented as a set of methods in classes. We reuse testing results of lower level interactions to compose testing results of higher level interactions.

In the code phase, Uzuncaova et al. [140] used extra specifications associated with features to generate a reduced test case set through a composition process to include more features. The extra specifications are represented with formulae in Alloy [68], embedded inside code and associated with features in an SPL. The argument is that large composed formulas are hard to solve. In addition, it is also difficult to generate test cases which correspond to models of these formulas; a small formula, plus a number of assumptions, is easier to use to construct a set of test cases. Uzuncaova et al. [140] reused a k-way partial product's test suite to generate a smaller size test suite for a (k+1)-way partial product. This is similar to our constraints-sensitive coverage in Chapter 3 because they considered constraints incrementally. Compared with our integration testing method in Chapter 5, they do consider integration testing for partial products, but they only consider one product at a time. That means that intermediate results of a k-way partial product are not saved and shared for another similar product, including the same k-way partial product. We explicitly capture such inclusion relations systematically with the interaction tree hierarchy/inclusion-graph, and then share results for all products. The reuse over a complete SPL is an advantage of our technique.

Reis et al. [114] applied integration testing over an SPL with a testing model, which is a UML 2.0 activity diagram. The objective was to develop an optimal set of paths to cover all 2-way interactions (edges) in an activity diagram. Their contribution is to abstract away a variation point with a set of variants in an activity diagram with an abstract node. With such a simplicification, they can apply Wang et al.'s [145] method to generate an optimal set of paths to cover all 2-way directed interactions in a normal activity diagram without variability nodes. Compared with our work in Chapter 5, there are several differences: 1) we use a code level representation, and they use an activity diagram; 2) we target all-way interactions including 2-way, and they only target 2-way interactions; and 3) our composed summaries (more specifcially the path conditions part) can directly be used

to generate test cases to trigger these interactions, and they only generate a set of paths which includes all features, but these paths can not be used directly to generate test cases.

In 2010, Kim et al.[79] introduced a similar reduction for a given monitoring property. In 2011, Kim et al [78] also found that for a given test case there are only a subset of products related to such test case. They developed a static analysis technique to define the meaning of *related features* as those which are reachable and which potentially change the data flow or control flow of one of the related features. The base case is that immediatly reachable features are related features. Compared with our method in Chapter 5, we have different testing targets. We target all interactions efficiently by reusing testing results among them, and they target running a test case/monitor for a reduced number of products.

In the architecture phase, Fischbein et al. [44] extended existing behavioral models, Labelled Transition Systems [85] and Modal Transition System, to support the conformance checking over SPLs. That is, such testing can check if a product is a child of an SPL based on the fact that all behaviors of such product are included in all possible behaviors of all products of such SPL. In 2010 Classen et al. [18] applied model checking over a feature-extended transition systems (FTS), a behavior model for SPLs. In 2011 they [17] continued to propose a symbolic model checking method to tackle the state explosion for all products of an SPL. Compared with our method in Chapter 5, there are two differences: 1) they are targeting the models and we are targeting the code representation of an SPL; and 2) they verified a property over an SPL, and we target all directed interactions in an SPL.

In summary, we introduced software product lines with a focus on feature models, then illustrated covering arrays and constraint solving as the background for our sampling techniques, and finally we presented the symbolic execution as the background for our compositional symbolic execution technique. We also discussed recent work on testing

SPLs that addresses coverage criteria and exploiting of similarities. Next we introduce the coverage criteria focusing on the most important property of SPLs, variability.

Chapter 3

Coverage Criteria Related to Constraints and Interactions[1]

In this chapter, we introduce our work, *Coverage and Adequacy in Software Product Line Testing* [25], to setup a series of coverage criteria related to constraints and interactions. Recall that this is the first step to resolve the challenge of testing an SPL.

The input is a feature model of an SPL, which defines a scope of valid products. There are three steps involved in generating a sample based on a feature model with constraints: 1) automatically translate a feature model to a relation model, 2) map the relation model to a CCIT model, and 3) automatically generate a sample to fulfill coverage criteria. The first two steps are the focus, and the final step is discussed in Chapter 4, where we discuss the development of four variant techniques to handle the sample generation problem focusing on constraints.

Although our coverage is general for any feature model, we propose a concrete translation from one of the feature models, OVM, to illustrate the feasibility of our method. Next we will discuss the first two steps over OVMs.

[1]Part of this work has been published in [25]

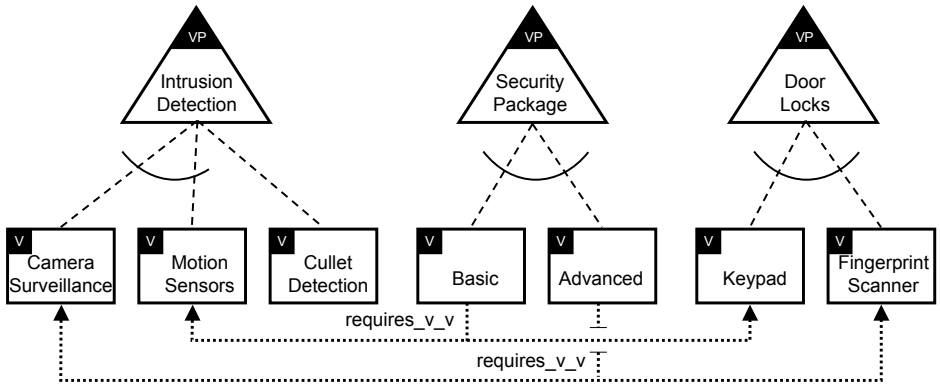


Figure 3.1: Example OVM Model[109]

3.1 Translating OVMs to Relation Models

We describe a relational model with two elements, a set of domains (D) with a finite set of values and a Cartesian product $\prod_{i=1}^{k} D_i$, where $k = |D|$. A Feature Model (FM) defines a set of valid products with respect to implicit and explicit constraints. We use a relational model equal to a feature model in terms of describing the same set of valid products. We construct such a relational model by re-describing atomic constructs of OVMs with these two elements. There are several key constructs in OVMs, including variation points, variants, and several dependencies including parent-children, mandatory, optional and alternative dependencies. We provide the translation next.

3.1.1 Translating Constructs

Variation points are mapped to domains, and variants are mapped to values. For the parent-children relations, we map values to corresponding domains. For example, for a feature model in Figure 3.1, we map a variation point, *door locks*, to a domain and two variants, *keypad* and *fingerprint scanner*, to two values. By defining the domain to contain these two values, we build a map of the parent-children relations.

A mandatory dependence states that a variant must be bound to a variant point in every product of an SPL. We can add such dependency relation after we construct a relation model for the remaining dependencies.

An optional dependence requires a variation point with a set of variants, and each of the variants may be bound to the variant point in a product of an SPL. We introduce a domain for each variant with two values, the variant itself and an empty value, denoted $\oslash$. We define $f(vp)$ to connect this set of domains to the domain of the variation point, vp. If we assume there are only optional relations between door-lock with keypad and fingerprint-scanner without the curve, then we have two domains for door-lock. One is $D_{door-lock_1} = \{keypad, \oslash\}$, and another is $D_{door-lock_2} = \{fingerprint - scanner, \oslash\}$. With a Cartesian product from these two domains, we can construct four possible partial products for door-lock: {keypad,fingerprint-scanner}, {keypad,$\oslash$}, {$\oslash$,fingerprint-scanner} and {$\oslash$,$\oslash$}. These four bindings satisfy the semantic of optional dependence relation as we described at the beginning of this paragraph.

An alternative dependence requires that a set of variants is bound to a variation point and associated with a bound $[i, j]$, where $i \leq j$. The bound means that the variation point must be bound to at least i number and at most j number of distinct variants. There is a special bound $[1, 1]$, which means there is exactly one feature chosen from a set of variants of a variation point in any product of an SPL. We map an alternative dependence relation to j number of domains. The first i number of domains has a set of values, all variants of the variation point. The remaining $j - i$ number of domains has another set of values, all variants with an extra value $\oslash$. Furthermore, we express the *distinct* semantic meaning with the inequality among the bound values.

For example, Figure 3.2 shows two alternative dependencies. The first has a bound $[1, 2]$. It is transferred to two domains, $D_{OneOrTwo_1} = \{A, B, C\}$ and $D_{OneOrTwo_2} = \{A, B, C, \oslash\}$. We need an inequality constraint between these two domains, and we

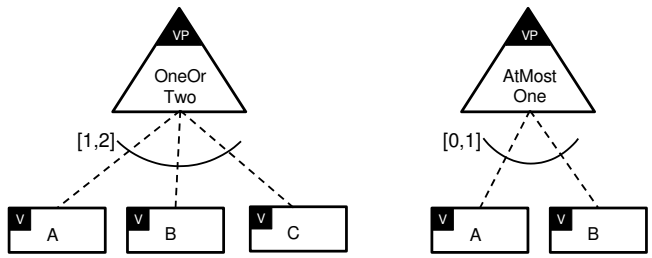


Figure 3.2: Alternative Choice Examples

delay the discussion with other explicit constraints together in Section 3.1.2. The second is special because of the lower bound expression $i = 0$. Such zero lower bound infers that we can define domains with $\oslash$. In this example, we define $D_{AtMostOne} = \{A, B, \oslash\}$. With only one domain, there is no need for introducing equality constraints.

Heretofore, we introduce both semantic meanings and relations of different dependencies. Next we translate constraints to relations.

3.1.2 Translating Constraints

We define a set of products without constraints as a relation shown below:

$$U = \Pi_{vp \in OVM} \Pi_{f \in f(vp)} D_f$$

Below we introduce several types of constraints to reduce the size of U incrementally. An inequality constraint between factors i and j is defined as:

$$I(i,j) \quad = \quad \{t \mid t \in U \wedge (\pi(t,i) \neq \oslash \Rightarrow \pi(t,i) \neq \pi(t,j))\}$$

The π is an extract function with two inputs, a tuple and an index, and one output, the corresponding value of that tuple for that index. For example, for a product, $t = (A, B, C)$, the $\pi(t, 0) = A$ and $\pi(t, 2) = C$.

The cumulative inequality constraint for a variation point, vp, is

$$I(vp) = \bigcap_{i \in f(vp), j \in f(vp) - \{i\}} I(i, j)$$

and for an OVM model:

$$I = \bigcap_{vp \in OVM} I(vp)$$

Other than the inequality constraint we introduce during the translation process, explicit constraints in OVM have several types: variant to variant (v_v), variation point to variation point (vp_vp) and variation point to variant (vp_v).

For v_v, there are two types of constrains as *requires* and *excludes* between two values v and w for two corresponding variation points, i and j. A translation for *requires* is shown below:

$$\begin{aligned} R(i, v, j, w) \quad = \quad & \{t \mid t \in U \wedge (\exists_{f \in f(i)} : \pi(t, f) = v) \wedge \\ & (\exists_{f \in f(j)} : \pi(t, f) = w)\} \cup \\ & \{t \mid t \in U \wedge (\forall_{f \in f(i)} : \pi(t, f) \neq v)\} \end{aligned}$$

The *excludes* is shown below:

$$\begin{aligned} E(i,v,j,w) &= \{t \mid t \in U \wedge (\exists_{f \in f(i)} : \pi(t,f) = v) \wedge \\ &\quad (\forall_{f \in f(j)} : \pi(t,f) \neq w)\} \cup \\ &\quad \{t \mid t \in U \wedge (\forall_{f \in f(i)} : \pi(t,f) \neq v)\} \end{aligned}$$

For vp_vp, we only list the transformation of *requires* constraints below:

$$\begin{aligned} R(i,j) &= \{t \mid t \in U \wedge (\exists_{f \in f(i)} : \pi(t,f) \neq \oslash) \wedge \\ &\quad (\exists_{f \in f(j)} : \pi(t,f) \neq \oslash)\} \cup \\ &\quad \{t \mid t \in U \wedge (\forall_{f \in f(i)} : \pi(t,f) = \oslash)\} \end{aligned}$$

Because of similarity among the constraint translations, we omit other types of constraints. Based on the above translations, we present a complete relation model for a feature model. The final model, FU, is a conjunction of all constraints for an OVM model:

$$FU = U \cap I \cap \bigcap R(\ldots) \cap \bigcap E(\ldots)$$

FU defines the same set of *valid* products compared with a feature model, and it is easier to be translated to a CCIT model for the next task, defining coverage criteria in terms of interactions and constraints.

3.2 CCIT Models and SPL Test Coverage Criteria

The mapping from a relation model to a CCIT model is straightforward. A domain of a relation model is mapped to a factor. The cardinality of a domain corresponds to

the choices of the factor, and then the constraints are rewritten with factor and value bindings. With a mapped CCIT model, we define two families of coverage criteria: an interaction-strength related criteria and a constraint-sensitive related criteria.

For the interaction-strength coverage criteria, we can manipulate the strength t to construct a series of coverage criteria. A covering array with a strength t is a subset of products of FU, expressed as $N \subseteq FU$. We further encode the strength on the top of FU. Let $F = \bigcup_{vp \in OVM} f(vp)$ be the set of all factors in a relational model. For all $S \subseteq F \wedge |S| = t$ let $RT_S = \Pi_{f \in S} D_f$ be the indexed set of all pairs of values over the t-size subset of factors, S. For N to be a t-way covering array, it must be the case that

$$\forall_{S \subseteq F \wedge |S| = t} : \forall_{t_s \in RT_S} : \exists_{t \in N} \pi(t, S) = t_s$$

Informally, this means that all possible t-sized tuples must be embedded in some full-tuple in N. When we increase t from 2 to n, where n is the maximum length among valid products, we increase the size of RT_S, which in turn increases the size of a covering array.

For the constraint-sensitive coverage criteria, we exploit the number of constraints incrementally. The base case is to apply no constraints, and the set of products corresponds to U. We incrementally add constraints to reduce the set size from U to a real valid product set. The reason for proposing this series of coverage criteria is that there are few covering array generators to handle constraints efficiently. Although we developed several cost-effective generators, the constraints problem inherently is a NP-Hard problem, so we still recognize these coverages as effective means for controlling the cost of test case generation activity. There is also a cumulative test coverage and corresponding targeted testing strategy.

In summary, we introduced an OVM translation with a simple relational language, and then transformed the relation model to a covering array model. Then we utilized a CCIT

model to explicitly express a series of interaction-strength related and constraint-sensitive coverage criteria. Next we introduce extensions of AETG to generate a covering array to fulfill these coverage criteria.

Chapter 4

Sampling Technique Focusing on Constraints[1]

We developed four variants of AETG to handle constraints with an incrementally deeper integration between AETG and SAT solvers. These variants are AETG-SAT [27], AETG-History [26], AETG-Threshold [28] and AETG-Hist-Threshold [28]. Due to the similarity of these algorithms, we focus on AETG-SAT, AETG-History and AETG-Threshold in detail in this chapter. For AETG-SAT, we introduce how to integrate a row construction process with a SAT checking process. For AETG-History, we introduce a deeper integration, which exploits a model in a SAT solver to infer *must* and *may* information and then to speed up an AETG row construction process. For AETG-Threshold, we use a model in a SAT solver directly to replace the row constructions of AETG at a threadhold point. Finally, we compare these four variants with AETG to show the effectiveness with our empirical study, which includes four 4 subjects and 30 synthesized CCIT models. Next we begin with an introduction of the base algorithm, AETG.

[1]Part of this work has been published in [27, 26, 28]

4.1 Basic AETG

Many algorithms and tools exist that construct covering arrays, but we focus in this dissertation on one-row-at-a-time greedy-algorithms in the style of the automatic efficient test case generator (AETG) [22]. Multiple variants of AETG have appeared in the literature, e.g., [11, 37, 29, 136]. We refer to these as *AETG-like*.

Algorithm 2 sketches the basic structure of this algorithm. Prior to execution an initialization step is used to calculate the number of t-sets for the given problem; covering all such sets drives continued execution of the algorithm. The algorithm constructs an array with numTests rows. A single row for the array is constructed in each iteration of the loop at line 4 until all t-sets have been covered. The algorithm constructs numCandidates different rows, line 5, and selects the best one to add to the array, lines 15-17. The choice of the size of candidate set is one of the differentiators of AETG-like algorithms. Our algorithm uses the value 50 for numCandidates to be consistent with the original description of AETG [22].

To build a single row, heuristics are applied to select the first factor and its value, lines 7-9. In AETG a factor-value pair is chosen that currently has the largest number of t-sets left to cover. The order in which the remaining factors are processed is randomly shuffled, line 10, and then the best value for each factor is selected, line 12-13, where the best value produces the most previously uncovered t-sets. In each step where a "best" decision is made, as well as where the first factor and value is selected in lines 7-9, ties are broken randomly, causing non-determinism in differing runs of the algorithm. Other greedy algorithms [29, 136] use slightly different heuristics to select the factor ordering.

```
mAETG(CAModel)
Require: uncovered-t-set-count: calculated by initialization
 1: numCandidates = 50
 2: numTests = 0
 3: testCasePool = ∅
 4: while uncovered-t-set-count > 0 do
 5:     for count = 1 to numCandidates do
 6:         testCase_count=generateEmptyTestCase()
 7:         l=selectFirstFactorValue(unCovSet)
 8:         f=selectFirstFactor(l)
 9:         insertValueForFactor(l,f,testCase_count)
10:         p=permuteRemainingFactors()
11:         for f ∈ p do
12:             l=selectBestValue(f)
13:             insertValueForFactor(l,f,testCase_count)
14:         saveCandidate(testCasePool,testCase_count)
15:     selectBestCandidate(testCasePool)
16:     update(uncovered-t-set-count)
17:     increment numTests
```

Algorithm 2: AETG Algorithm

4.2 AETG with Basic SAT Checking

In this section, we first introduce how to express constraints of CCIT models with a boolean formula, and then discuss the AETG-SAT algorithm in detail.

4.2.1 Translating Constraints as Boolean Formulae

We use a boolean propositional formula as the logic language to express constraints in a CCIT model, which in turn comes from a feature model of an SPL. Boolean propositional logic is the basis for almost all other higher level logic languages, such as variant decision procedures [82], Alloy [68] and other logic languages with targeted application domains. We choose the basic logic for two reasons: 1) it is straightforward to represent both the *excludes* constraints and the partial bindings during a row construction in AETG compactly; and 2) there are an abundance of tools implemented with the C language, and our tool is developed with C.

We give an example shown in Figure 4.1 to show exactly how to express constraints

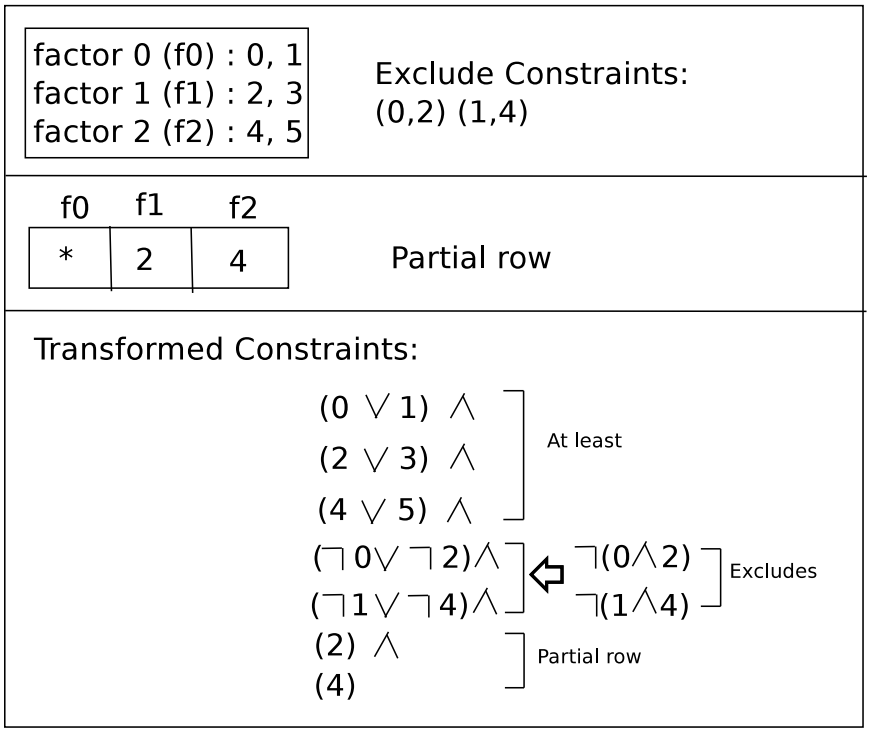


Figure 4.1: A propositional formula for a CIT with AETG construction

with boolean logic. We can see there are three factors, each of which has two values. There are two *excludes* constraints. The first constraint states that if Factor *fo* is bound to the value 0, then Factor *f1* can not be bound to value 2. During a covering array construction, we have a partial row shown in this example, which states that the factor *f1* and *f4* are bound to 2 and 4 correspondingly. Under this scenario, a basic question is to ask if such binding is satisfiable. That is, if there exists a binding for the factor *fo* such that the binding together with the partial binding does not conflict with the *excludes* constraints. We translate the whole question to a formula shown in the figure. There are three parts. The first part is called *at least*, which guarantees that for each factor, there must be a binding. The second part corresponds to the *excludes* constraints. The format can be transformed from a conjunctive form to a disjunctive form, which is just a concrete format for underlying SAT solvers. Finally, the third part is the partial bindings. From

top to bottom, all clauses are connected with conjunctive boolean operations, and the whole formula follows the conjunctive normal form (CNF) format.

After we feed this formula to a SAT solver, there should be an UNSAT answer. We can see under such partial row that there is no value for *f0* such that a complete row does not violate the constraints. For example, if we bind 0 to *f0*, then (0,2) can not be together; if we bind 1 to *f0*, then (1,4) can not be together. So we say such partial row is not satisfiable, and we need to change a bounded value for *f1* or *f2*. Fortunately, all such reasoning automatically happens in a SAT solver. Next we introduce how the AETG algorithm exactly integrates a SAT solver.

4.2.2 SAT Checking

Algorithm 3 describes the AETG algorithm with constraint checking. Recall that AETG is to construct a covering array row by row. For each row we fill in the row factor by factor, and for each factor we try every value one by one. The core of an integration between AETG and SAT solver is to check if a partial row is satisfiable by avoiding a conflict with constraints. More precisely, whenever a factor is bound with a value, at that moment, we check if such binding is possible. We examine decomposed steps in the algorithm to describe the integration in more detail.

Algorithm 3 can be roughly partitioned into three parts. The first part is the *Requires* statement; the second part corresponds to 4-14, and the third part corresponds to the statement 15. All other statements are supporting operations for making transitions smoothly among these three parts.

The *requires* statement sets up the number of value combinations to be covered. With the example shown in Figure 4.1, if we setup the strength as 2, then the requires statement lists all covered pairs as {(0,2),(0,3),(0,4),(0,5),(1,2),(1,3),(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)}.

```
mAETG-SAT(CAModel)
Require: uncovered-t-set-count: calculated by initialization
 1: numCandidates = 50
 2: numTests = 0
 3: testCasePool = ∅
 4: while uncovered-t-set-count > 0 do
 5:     for count = 1 to numCandidates do
 6:         testCase_count=generateEmptyTestCase()
 6a:        sat=false
 6b:        while !sat
 7:             l=selectFirstFactorValue(unCovSet)
 8:             f=selectFirstFactor(l)
 8a:            sat=¬factorInvolved(f) ∨ checkSAT(testCase_count)
 9:         insertValueForFactor(l,f,testCase_1)
10:         p=permuteRemainingFactors()
11:         for f ∈ p do
11a:            sat=false
11b:            tries = 1, maxTries = v
11c:            while !sat and tries ≤ maxTries
12:               l=selectBestValue(f)
12a:              sat=¬ factorInvolved(f) ∨ checkSAT(testCase_count)
12b:              increment tries
13:             insertValueForFactor(l,f,testCase_count)
14:         saveCandidate(TestCasePool,testCase_count)
15:     selectBestCandidate(testCasePool)
16:     update(uncovered-t-set-count)
17:     increment numTests
```

Algorithm 3: AETG-SAT Algorithm

Furthermore, for constraints listed in the figure, we need to check if each pair is satisfiable. So the *requires* statement involves two steps: an encoding for pair bindings and a satisfiable checking, which are not shown in the algorithm.

The second part is to construct rows one by one. It can be further partitioned to 6b-9, 10-13 and other supporting statements. For 6b-9, they are used to choose the first factor and to bind a value for this factor. Such operations are from a traditional AETG procedure. The most interesting statement is 8a, which is integrated with a SAT solving operation to check if a partial binding is satisfiable with the function called checkSAT($testCase_{count}$). The $testCase_{count}$ looks like the partial row shown in Figure 4.1. After we bind a value for the first factor, 10-13 are used to bind values for other factors one by one in a random

order. We can see that Statement 12a has the same constraint checking call as Statement 8a.

After the second part is finished, we constructed 50 rows. In statement 15, we choose the best row, which covers most uncovered value combinations, as a row in the final covering array. These three steps guarantee that a covering array is consistent with constraints. We observed that there is a potential opportunity to exploit the information in SAT solvers to save row-construction efforts in AETG.

4.3 AETG With SAT History

The information we can get from a SAT solver is a complete row based on a partial input row from AETG. The row satisfies all constraints. We partition this returned complete row ino two parts, the Must/May and others. Figure 4.2 shows a complete row corresponding to the SAT path in the search tree. We can see that such a search begins when we feed a partial row which has only the first factor f bound with $v1$. Based on the BCP stage of a SAT search procedure, from top down we may infer other bindings. All inferred bindings must happen based on constraints of a CCIT model and the initial binding. From these bindings in Figure 4.2, there are negative and positive values. For positive values, we call these as *Must* information, because these bindings will be used in AETG for filling in the corresponding factor without any choice. With the same example, $x10$ is a *must* information for the factor i. For negative values, we call these as *MustNot* information, because these bindings will be used in AETG so that AETG will not use these values for corresponding factors. If we apply the original value set of a factor to subtract *MustNot*, then the remaining values are called *May* set. For simplicity, we call these negative values as *May* values. For example, we have $!x6$ in the figure, and this MustNot leads to an *May* including {v7,v8} for Factor h. We call these *Must/May* bindings as the history

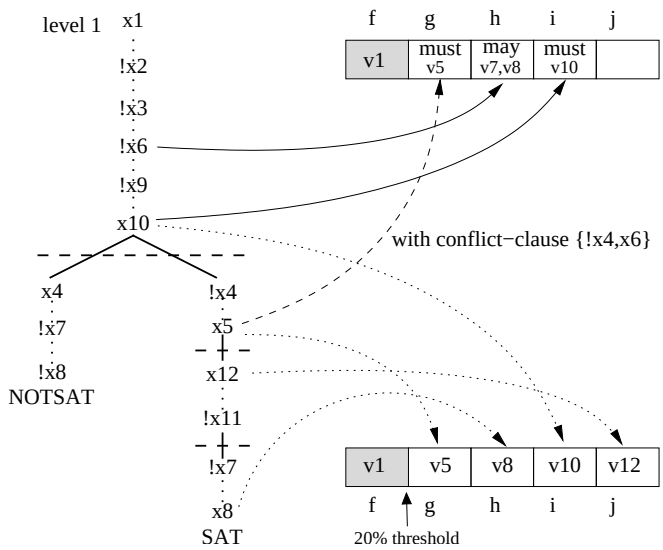


Figure 4.2: The Exploitation Based on SAT History and Threshold

information. Next we use an algorithmic representation to describe the *Must* and *May* exploitation in more detail.

Algorithm 4 integrates such an exploitation into Algorithm 3. For statements 6b-9 in Algorithm 3, there is no change between these two versions. This is reasonable because for the first factor binding, we do not have any history information. For statements 10-13 in Algorithm 3, there is a dramatic change for exploiting the history. Statement 12 is changed to selelctBestValueFromMaySet(f,maySet) from the original selectBestValue(f). This statement is to choose a best value for a factor, f. With the history, we choose values only in the *May* set, which can save some choosing efforts of AETG. The *May* set is calculated from the added statement 11d, which uses the returned *MustNot* information from SAT solvers to compute *May*. For Statement 13, there are four more added sub-statements. The core one is Statement 13a, which computes the *Must* information by copying the *Must* information from SAT solvers. The remaining three statements from 13b-13d apply these *Must*s to factor bindings and help Statement 11 to skip these factors during the operation to choose the next factor.

```
mAETG-History(CAModel)
Require: uncovered-t-set-count: calculated by initialization
 1: numCandidates = 50
 2: numTests = 0
 3: testCasePool = ∅
 4: while uncovered-t-set-count > 0 do
 5:     for count = 1 to numCandidates do
 6:         testCase_count=generateEmptyTestCase()
 6a:        sat=false
 6b:        while !sat
 7:             l=selectFirstFactorValue(unCovSet)
 8:             f=selectFirstFactor(l)
 8a:            sat=¬ factorInvolved(f) ∨ checkSAT(testCase_count)
 9:         insertValueForFactor(l,f,testCase_count)
10:         p=permuteRemainingFactors()
11:         for f ∈ p do
11a:            sat=false
11b:            tries = 1, maxTries = v
11c:            while !sat and tries ≤ maxTries
11d:                maySet=mineMayAssignments()
12:                 l=selectBestValueFromMaySet(f,maySet)
12a:                sat=¬ factorInvolved(f)∨ checkSAT(testCase_count)
12b:                increment tries
13:             insertValueForFactor(l,f,testCase_count)
13a:            mustSet=mineMustAssignments()
13b:            for(l, f) ∈ mustSet do
13c:                insertValueForFactor(l,f,testCase_count)
13d:                p = p − f
14:         saveCandidate(TestCasePool,testCase_count)
15:     selectBestCandidate(testCasePool)
16:     update(uncovered-t-set-count)
17:     increment numTests
```

Algorithm 4: AETG-History Algorithm

Based on the above discussion, it is possible to be more ambitious by using *full* bindings instead of only the *Must/May* history information in order to accelerate row-constructions of AETG faster. We discuss such an optimization next.

4.4 Threshold Triggered SAT Assignments

The presence of constraints tends to reduce the size of the valid solution space. As a row is built, this may lead to an increasingly limited set of valid choices of factor-values,

especially late in the row. An expensive portion of an AETG-like algorithm is the method selectBestValue, line 12, which requires a linear scan of each possible value for the current factor. For each value it requires $\binom{j}{t}$ evaluations to compute how many new t-sets will be covered by that choice, where j is the current loop iteration starting at line 11. This requires a total of $v \times \binom{j}{t}$ computations for each call of this method. In the constrained portion of the search space, which lies near the end of a row, the cost of this scan may yield little benefit since few consistent values may remain for a factor.

When the SAT solver finds a satisfying assignment it calculates a complete configuration. That configuration may not, however, be one that drives the overall CCIT solution to a small CMCA – this is the intent of the AETG heuristics. The time needed to generate a CMCA can be reduced by short-circuiting the AETG calculations in lines 11-13 using the assignment calculated by the most recent successful SAT call. Figure 4.2 illustrates this process when only one out of five, or 20%, of the factors is assigned. The remaining four factor-value bindings are extracted from the satisfying assignment – illustrated by dotted arrows – and used to complete the row.

Short-circuiting AETG calculations early in a row can speedup solution times, but this may lead to larger CMCAs. Waiting until 100% of the factors are assigned yields no performance improvement, but also no increase in CMCA size. For algorithmic frameworks like this it is necessary to identify the parameter value that provides a desirable cost-benefit tradeoff. We refer to this parameter as the row *threshold* and discuss finding a good value for the threshold in Section 5.5.

Algorithm 5 presents the AETG-Threshold algorithm. It differs from Algorithm 3 (AETG-SAT) only after the threshold has been reached. In the initialization step the threshold value is input as a percentage of the row size, and translated into the threshold index; the switching point in this algorithm. Then in step 11, at 11aa, a check is made to determine if this threshold has been reached. If it has not, the algorithm continues as

```
mAETG-Threshold(CAModel)
Require: uncovered-t-set-count, thresholdIndex: calculated by initialization
 1: numCandidates = 50
 2: numTests = 0
 3: testCasePool = ∅
 4: while uncovered-t-set-count > 0 do
 5:     for count = 1 to numCandidates do
 6:         testCase_count=generateEmptyTestCase()
 6a:        sat=false
 6b:        while !sat
 7:             l=selectFirstFactorValue(unCovSet)
 8:             f=selectFirstFactor(l)
 8a:            sat=¬ factorInvolved(f) ∨ checkSAT(testCase_count)
 9:         insertValueForFactor(l,f,testCase_count)
10:         p=permuteRemainingFactors()
10a:        index=1
11:         for f ∈ p do
11aa:           if index < thresholdIndex
11a:            sat=false
11b:            tries = 1, maxTries = v
11c:            while !sat and tries ≤ maxTries
12:             l=selectBestValue(f)
12a:            sat=¬ factorInvolved(f)∨ checkSAT(testCase_count)
12b:            increment tries
13:             insertValueForFactor(l,f,testCase_count)
13a:            increment index
13b:          else
13c:            satAssignedSet=mineRemainingAssignments()
13d:            for(l, f) ∈ satAssignedSet do
13e:              insertValueForFactor(l,f,testCase_count)
14:         saveCandidate(TestCasePool,testCase_count)
15:     selectBestCandidate(testCasePool)
16:     update(uncovered-t-set-count)
17:     increment numTests
```

Algorithm 5: AETG-Threshold Algorithm

normal, allowing AETG to select the best symbol, followed by consistency checks using the SAT solver. Once the threshold value has been reached, execution switches, and we mine the SAT assignment from the most recent SAT call (mineRemainingAssingments) and save this as the satAssignedSet. The entire assignment is then used to fill in the remaining factor-values without the use of the AETG strategy in lines 13b-13e. We note that the SAT solver makes random decisions at points in its search that are independent from that of the AETG-like algorithm.

4.5 Combining History and Threshold Optimizations

The history and threshold optimizations both seek to fill in multiple factor-value bindings in a single step. The advantage of AETG-History is that is guaranteed to not interfere with AETG heuristics and, consequently, will not increase CMCA size as is possible with AETG-Threshold. On the other hand, since AETG-History generally only fills in a portion of the row, it will not reduce solution time as much as AETG-Threshold.

We consider a simple combination of these two algorithms which we call AETG-Hist-Threshold. The algorithm is not shown since it is straightforward variation of Algorithm 4, which is used as the base algorithm up until a threshold has been reached for each row. After the threshold is reached the algorithm switches strategy and mines the current SAT assignment to fill in the remaining factor-values as in Algorithm 5.

Next we introduce our experimental data to illustrate the effectiveness of these 4 variants, AETG-SAT, AETG-History, AETG-Threshold and AETG-Hist-Threshold.

4.6 Empirical Investigation

We begin our empirical investigation by summarizing five case studies based on four software subjects. Note for evaluting the effectivness and efficiency of CCIT aglorithms, we can use broader range of subjects than SPLs, which have a CIT model with a rich set of constraints. In this study, we use highly-configurable software systems as our subjects. These show the abundance and types of constraints found in real software systems. We then present an analysis designed to evaluate the performance of four algorithms presented above with respect to generation time and sample size. We utilize the five case studies and generate an additional set of synthesized CCIT problems for this analysis.

4.6.1 Case Studies

We have chosen four non-trivial highly-configurable software systems – SPIN [65], GCC [46], Apache [1] and Bugzilla [96] to study with respect to constraints. We analyzed the configuration options for these tools based on available documentation and constructed models of the options and any constraints among those options. All of our models should be considered an approximation of the true configuration space of the programs. One way we do this is by ignoring options we regard as overlapping, i.e., an option whose only purpose is to configure another set of options is ignored, as well as options that serve only to define program inputs. Another is by underestimating the number of possible values for each option. If an option takes an integer value in a certain range we apply a kind of category partitioning and select a default value, a non-default legal value, and an illegal value; clearly one could use more values to explore boundary values, but we choose not to do that. Similarly for string options we choose values modeling no string given, an empty string, and a legal string. Ultimately, the specific values chosen are determined during test input generation for a configuration, a topic we do not consider here. We report data on the size of these models, the number and variety of constraints, and the existence of implied forbidden t-sets.

4.6.1.1 SPIN Model Checker

SPIN is a widely-used publicly available model checking tool [65]. SPIN serves both as a stable tool that people use to analyze the design of a system they are developing, expressed in SPIN's Promela language, and as a vehicle for research on advanced model checking techniques; as such it has a large number and wide variety of options. We examined the manual pages for SPIN, available at [66], and used it as the primary means of determining options and constraints; in certain cases we looked at the source code

itself to confirm our understanding of constraints.

SPIN can be used in two different modes: as a *simulator* that animates a single run of the system description or as a *verifier* that exhaustively analyzes all possible runs of the described system. The "-a"options select verifier mode. The choice of mode also toggles between partitions of the remaining SPIN options, i.e., when simulator mode is selected the verifier options are inactive and vice-versa. While SPIN's simulator and verifier modes do share common code, we believe that the kind of bi-modal behavior of SPIN warrants the development of two configuration models – one for each mode.

The simulator configuration model is the simpler of the two. It consists of 18 factors and ignoring constraints it could be modeled as a $MCA(N; 2, 2^{13}4^5)$, i.e., 13 binary options and 5 options each with 4 different values; this describes a space of $8.3 * 10^6$ different system configurations. It has a total of 13 pairwise constraints that relate 9 of the 18 factors. The nature of the interactions among the constraints for this problem, however, give rise to no implied forbidden pairs. As for most problems, constraints for this problem can have a dramatic impact – enforcing just 1 of the 13 constraint eliminates over 2 million configurations.

The verifier configuration model is richer. It is worth noting that running a verification involves three steps. (1) A verifier implementation is *generated* by invoking the spin tool on a Promela input with selected command line parameters. (2) The verifier implementation is *compiled* by invoking a C compiler, for example gcc, with a number of compilation flags, e.g., "-DSAFETY", to control the capabilities that are included in the verifier executable. (3) Finally, the verifier is *executed* with the option of passing several parameters. We view the separation of these phases as an implementation artifact and our verifier configuration model coalesces all of the options for these phases. This has the important consequence of allowing our model to properly account for constraints between configuration options in different phases. The model consists of 55 factors and

ignoring constraints it could be modeled as a $MCA(N; 2, 2^{42}3^{2}4^{11})$; this describes a space of $1.7 * 10^{20}$ different configurations. This model includes a total of 49 constraints – 47 constraints that either require or forbid pairs of combinations of option values and 2 constraints over triples of such combinations. An example of a constraint is the illegality of compiling a verifier with the "-DSAFETY" flag and then executing the resultant verifier with the "-a" option to search for acceptance cycles; we note that these kinds of constraints are spread throughout software documentation and source code.

The set of SPIN verifier constraints span the majority of the factors in the model – 33 of the 55 factors are involved in constraints. Furthermore, the interaction of these constraints through the model gives rise to 9 implied forbidden pairs.

4.6.1.2 GCC Optimizer

GCC is a widely used compiler infra-structure that supports multiple input languages, e.g., C, C++, Fortran, Java, and Ada, and over 30 different target machine architectures. We analyzed version 4.1, the most recent release series of this large compiler infra-structure that has been under development for nearly twenty years. GCC is a very large system with over 100 developers contributing over the years and a steering committee consisting of 13 experts who strive to maintain its architectural integrity.

As was done for SPIN, we analyzed the documentation of GCC 4.1 [46] to determine the set of options and constraints among those options; in some cases we ran the tool with different option settings to determine their compatibility. We selected a core component of GCC, the machine-independent *optimizer*, and modeled it with 199 factors and 40 constraints.

The optimizer model, without constraints, can be modeled as a $MCA(N; 2, 2^{189}3^{10})$; this describes a space of $4.6 * 10^{61}$ different configurations. Of the 40 constraints, 3 are three-way and the remaining 37 are pairwise. These constraints are related to 35 of the

199 factors and their interaction gives rise to 2 implied forbidden pairs.

Examples of constraints on optimizer settings include: "-finline-functions-called-once ... Enabled if -funit-at-a-time is enabled." and "-fsched2-use-superblocks ... This only makes sense when scheduling after register allocation, i.e. with -fschedule-insns2". We took the following approach to defining constraints. The commonly used "-O" options are interpreted as option packages that specify an initial set of option settings, but which can be over-ridden by an explicit "-fno" command. Interpreting these more strictly gives rise to hundreds of constraints many of which are higher-order, i.e. they constrain three or more factor-values.

4.6.1.3 Apache HTTP Server 2.2

The Apache HTTP Server 2.2, is an open source, widely used, web server that works on both UNIX and Windows platforms. It can be customized by the system administrator through a set of *directives*. The directives for Apache fall into nine categories, which include the core program, extensions, server configuration, etc. In total there are 379 configurable options that contribute to these categories. For the purposes of our case study we initially limited our examination to the 166 options related to *h* directives from the user manual. Upon further examination, we found that several of the constraints on this set of options involved an additional 6 factors that were not part of the *h* directives. We added those options to our model for a total of 172 options. The final model has mostly binary options (92%) with a small number of factors that have between 3 and 6 options. The unconstrained Apache can be modeled as an $MCA(N; 2, 2^{158}3^{8}4^{4}5^{1}6^{1})$, i.e. there are 158 binary, 8 ternary, 4 four-valued, and one factor each that have five and six values. This leads to an unconstrained configuration size of $1.8*10^{55}$.

During our analysis, we uncovered 7 constraints in the Apache documentation that relate between 2 and 5 different options. An example of a constraint for Apache is that

the "Require" directive that selects which authenticated users can access a resource, must be accompanied by the "AuthName" and "AuthType" directives, as well as directives for "AuthUserFile" and "AuthGroupFile" (to define users and groups). Without these other directives being defined, "Requires" will not function properly. In total, only 18 options are involved in the 7 constraints. Of these constraints all but one are binary, and one is a ternary. There are no implicit 2-way constraints in this system.

4.6.1.4 Bugzilla 2.22.2

Bugzilla [96] is an open source defect tracking system from Mozilla. It provides developers with a mechanism to track outstanding bugs in their systems. The software includes advanced search capabilities, email notifications, multiple bug reporting formats, scheduled reports, time tracking, etc. It supports multiple database engines and is customizable by the user. After examining the documentation we selected three sections of the user manual to which we have restricted our analysis. These are the sections that contain the core functionality: Chapter 3. – *Administering Bugzilla*, Chapter 5. – *Using Bugzilla* and Chapter 6. – *Customizing Bugzilla*. Our analysis uncovered 44 options.

When conducting our analysis we found 10 additional options that were not included in one of the listed Chapters, but that were somehow related through constraints to options within the scope of our analysis; we added these into our model to be complete. Our final model has 52 factors of which 94.2% are binary. The final model for Bugzilla is a $MCA(N; 2, 2^{49}3^{1}4^{2})$. There are 49 binary, one ternary and 2 four-valued factors. This leads to an unconstrained configuration space of $2.7*10^{16}$. Bugzilla's documentation describes 5 constraints; 4 relating 2 options and 1 relating 3 options. An example of a Bugzilla constraint is when the "Mail Transfer Agent" is set to "Postfix", it requires that the "sendmailnow" option is turned on. In total, 11 options were involved in the 5 constraints. We did not uncover any 2-way implicit constraints for this system.

4.6.2 Synthesized CCIT Problems

The five case studies are essential elements of our evaluation, but they do not provide a large population of problems on which to compare algorithm performance. The time required to develop the case study models was significant and we felt that it was impractical to produce a significantly larger number of case studies in a timely fashion. Instead, we used the five case studies to develop a characterization of the abundance, type and complexity of constraints found in real systems and then used that characterization to synthesize a large number of CCIT problems to include in our evaluation.

In Table 4.1 and Table 4.2 we provide a summary of the CCIT models for the 5 case studies, highlighting their main characteristics. The table shows counts of the number of factors (Num Factor) and explicit constraints (Num Cons) for each problem. It also provides the number and percentage (in parentheses) of factors with 2, 3, 4, or more values. Similarly for constraints it provides the number and percentage (in parentheses) of constraints of arity 2, 3, or more. As discussed in Section 4.2.2 at Line 8a, it is possible to skip constraint processing during CMCA construction for factors that are not involved in constraints – the second column (Factor Invol.) under the Constraints sub-heading provides the number and percentage (in parentheses) of factors involved in constraints. The last two columns in the table show the dual of this information – they provide the number of constraints in which a factor participates. This provides an indication of the extent to which constraints are "coupled" and may give rise to implied constraints. For example, if a factor is involved in only a single constraint it will fall into the first (1 Cons. Per Factor) category. We do not show data for factors involved in more than 2 constraints due to space limitations.

We use the summarization of case study characteristics to synthesize random covering array models with constraints that share the characteristics of the case study systems.

	Factors and Values				
	Num Factor	2 Values	3 Values	4 Values	5 or 6 Values
$Spin_s$	**18**	13 (72.2)	0 (0.0)	5 (27.8)	0 (0.0)
$Spin_v$	**55**	42 (76.4)	2 (3.6)	11 (20.0)	0 (0.0)
GCC	**199**	189 (95.0)	10 (5.0)	0 (0.0)	0 (0.0)
Apa	**172**	158 (91.9)	7 (4.1)	4 (2.3)	2 (1.2)
Bugz	**52**	49 (94.2)	1 (1.9)	2 (3.8)	0 (0.0)

Table 4.1: Case Study Basic Characteristics: Factors and Values

	Constraints					Factor Involv.	
	Num Cons	Factor Invol.	2-way Cons	3-way Cons	4/5-way Cons	1 Con. Per Factor	2 Con. Per Factor
$Spin_s$	**13**	9 (50.0)	13 (100.0)	0 (0.0)	0 (0.0)	5 (55.6)	0 (0.0)
$Spin_v$	**49**	33 (60.0)	47 (95.9)	2 (4.1)	0 (0.0)	12 (36.4)	7 (21.2)
GCC	**40**	36 (18.1)	37 (92.5)	3 (7.5)	0 (0.0)	14 (38.9)	13 (36.1)
Apa	**7**	18 (10.5)	3 (37.5)	1 (12.5)	3 (37.5)	14 (77.8)	1 (5.6)
Bugz	**5**	11 (21.2)	4 (80.0)	1 (20.0)	0 (0.0)	11 (100.0)	0 (0.0)

Table 4.2: Case Study Characteristics: Number and Percent of Factors/Constraints

Our synthesis algorithm starts by randomly generating a number of factors between 18 and 199 – the range of factors found in our case studies. The case studies had between 72% and 95% of their factors with only two values; 90% of the factors across all of the studies were binary. We skewed the number of binary factors towards the average across all case studies by selecting between 85-95% of the number of factors to be binary and the rest to involve between 3 and 6 factors. We weighted the latter decision with a 40% probability that 3 will be chosen, and a 20% probability for the rest.

The ratio of constraints to factors in the case studies varied from 0.04 to 0.89, but this degree of variation leads to large numbers of models that bear no resemblance to the

case studies. We chose to generate constraints by using the range of actual constraints, between 5 and 49, found in the case studies. Between 37% and 100% of the constraints are binary in our case studies; 90% of the constraints across all of the studies were binary. As with binary factors, we skewed the number of binary constraints towards the average across case studies by selecting 80-100% of constraints per problem to be binary. The remaining constraints chosen as 3, 4 or 5-way with equal probability. We used a greedy synthesis approach, so at each decision point if all constraints are assigned to a category synthesis stops.

Another consideration that we tried to enforce is to make sure that between 40-100% of the factors involved in constraints are involved in only a single constraint while 10-20% of the factors are involved in two constraints; the latter range represents the skewing of factor involvement toward the average across all five case studies. Any constraints that are not bound to factors are configured to be involved with between 3 and 9 constraints with equal probability.

We automated this approach to generate CCIT problems. The CMCA models, numbers and arity of constraints for all 30 synthesized CCIT problems are shown in Table 4.4. For each of the case studies, and synthesized CCIT problems, it enumerates the factors for the CAModel and the constraints. This information is given in an abbreviated form that shows the numbers of factors with a given number of values in the form $\#values^{\#factors}$ and the number of constraints with a given arity in the form $arity^{\#constraints}$ (column No. Cons.).

4.6.3 Performance Evaluation

In this section, we compare the performance of AETG-History, AETG-Threshold and AETG-Hist-Threshold using an incremental SAT version of AETG-SAT as the baseline.

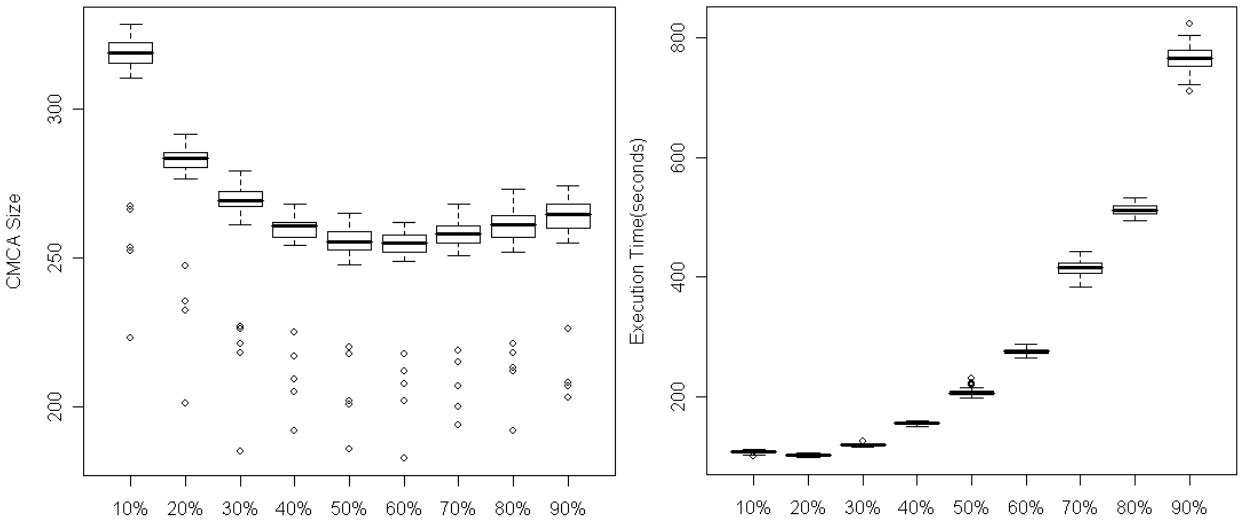


Figure 4.3: SAT Threshold Performance for 5 Random Samples

Our goal is to empirically evaluate the algorithms with respect to both the computational time required (efficiency) and the size (quality) of the resulting solutions. Before comparing AETG-History and AETG-Threshold, however, it is first necessary to select a *threshold value*. Our first study evaluates various threshold values to find the best balance of efficiency and quality in our data samples.

4.6.4 Finding a Good Threshold Point

The AETG-Threshold algorithm triggers a switch in the algorithmic behavior at a given threshold point. Once this threshold has been reached the algorithm stops any further AETG evaluations, and instead fills in all of the remaining factor-values using the last satisfying assignment found by the SAT solver. We expect a range of behavior for this algorithm. When the threshold is set very low we expect solutions that are effectively random in achieving interaction coverage, while thresholds closer to the end will likely save little computation. Since the selection of the threshold will affect our results, we compare both time and size over a range of threshold values. To determine a threshold we randomly selected 5 samples from our synthesized data. These are sample numbers

14, 15, 21, 28 and 29 from Table 4.4. We evaluate threshold in 10% increments from 10% through to 90%. For each threshold, we run AETG-Threshold 50 times and collect both the time in seconds as well as the size of the resulting CMCA. Initialization is performed once and this time is divided evenly among samples. Figure 4.3 shows box plots for time and size for each of the 50 runs summed across these 5 samples. The graph on the left plots size while the graph on the right plots execution time. These plots capture the variability in the 50 runs of the algorithm, while showing the median total times and sizes based on different threshold values.

As is expected the CMCA sizes for low thresholds are large compared with the sizes calculated for a threshold of 50% or greater. The run times are dramatically lower for thresholds below 50% and run time increases rapidly with increasing threshold.

Visual inspection of the plots, suggests that a threshold in the 50-70% range provides a good balance between speed and CMCA size. We are interested in confirming this intuition by calculating the threshold that provides the best cost-benefit tradeoff. We use a normalization technique to equalize the values for time and size to contribute equally in our decision. It is possible to associate more weight to one or the other of these metrics, depending on the final objective, but for our initial investigation we chose an equal contribution. Given the different scales, we reduce the impact of each to a relative importance. We first calculate the *timeRatio* and *sizeRatio* by subtracting the minimum time (or size) from the time (or size) of each threshold point. For instance in the *timeRatio* all times will subtract 103.0 as is seen in Table 4.3. We then divide this number by the range of times (or range of sizes). This gives us a number between zero and one for each ratio. Zero means that the time (or size) matches the minimum value for all thresholds and one means it matches the maximum time (or size). We use a weighted sum of these, *combinedTimeSize*, and select the minimum value to set our best threshold. Since the ranges of data for time and size vary by a factor of *sizeRange*/*timeRange* (10.7 in our

	Threshold Percentage								
	10%	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**
Time	109.33	103.04	120.20	155.59	207.94	276.61	415.37	512.97	765.69
timeRatio	0.01	0.00	0.03	0.08	0.16	0.26	0.47	0.62	1.00
Size	312.64	278.64	264.56	255.60	251.22	250.42	253.62	256.94	259.96
sizeRatio	1.00	0.45	0.23	0.08	0.01	0.00	0.05	0.10	0.15
combinedTimeSize	10.66	4.83	2.45	0.97	0.30	**0.26**	1.02	1.73	2.63

Table 4.3: Time and Size of 5 Samples for Threshold Percentages

data set given a time range of 662.7 and a size range of 62.2), we multiply the size ratio by this value giving us the following formula

$$combinedTimeSize =$$

$$timeRatio + (sizeRange/timeRange) \times sizeRatio.$$

The data for these calculations for the 5 samples is given in Table 4.3. We use the average of the sum of the 50 data repeats as a basis for this calculation. The results of this analysis show that the threshold value of 60% provides the best balance of both time and size. In Table 4.4 and Table 4.5 we present only the data for this threshold value.

4.6.5 Comparing Algorithms

We compare the four variations AETG-SAT, AETG-Hist, with a threshold of 60% for AETG-Threshold and AETG-Hist-Threshold, and the unconstrained AETG algorithm. All of our implementations are written in C++ and use miniSAT v1.14.1 written in C [94]. All program run-time data is gathered by executing implementations on an Opteron 250 processor with 4 Gigabytes of RAM running the Fedora Core 3 installation of linux.

For each technique and CCIT problem, we ran 50 trials; this helps account to the random variation that is inherent in AETG-like algorithms. We collect both CMCA size and execution times for each of the 50 trials. Once again, we divide the initialization

times evenly among all runs. Table 4.4 and Table 4.5 correspondingly show the results of time and size of generating samples for $t = 2$ for the five case studies as well as the 30 synthesized CCIT problems. The first three columns of both tables identify the CCIT problem and characterize the CAModel and problem constraints. In Table 4.4, the remaining columns are split into a group of five, reporting for each of the five techniques in terms of execution time in seconds; in Table 4.5, the remaining columns are split into a group of five, reporting for each of the five techniques in terms of CMCA size. Each cell in both tables gives the average over the 50 trials for either size or time. The last row of both tables is the sum of averages across all data sets.

The variation in covering array size across techniques is relatively modest. It is noteworthy, that less than 3% of the MCA rows produced by the unconstrained AETG technique satisfy constraints. AETG-SAT and AETG-History produce nearly identical sizes as do AETG-Threshold and AETG-Hist-Threshold. A difference of 3 rows between AETG-Threshold and AETG-Hist-Threshold across the more than 1500 is within the expected variation attributable to randomization in AETG; as can be observed by the range in the box plots of the CMCA size data in Figure 4.3. The 60% Threshold algorithms appear to provide a modest reduction, approximately 3%, in CMCA size. We conjecture that the effectively random selection made after the threshold point provides a relaxation in the aggressive one-row-at-a-time greedy technique allowing better decisions to be made in later rows. We know that meta-heuristic search techniques that construct the entire array at a time (rather than fix one-row-at-time), and relax intermediate solutions by allowing occasional "bad choices", in general produce smaller covering arrays for both unconstrained and constrained CIT problems [23, 27]. Further analysis is needed to confirm this conjecture.

The variation in execution time data across techniques is more significant. AETG-History yields a 9% reduction in solution time over AETG-SAT and a 5% reduction in

solution time over unconstrained AETG. AETG-Threshold, as expected, significantly speeds up solution time by skipping AETG processing on 40% of each row; it yields a 67% reduction in solution time relative to AETG-SAT.

The AETG-Hist-Threshold shows that the History and Threshold optimizations target different aspects of AETG-SAT algorithm since the data reveal that their benefits accumulate when the optimizations are composed. The AETG-Hist-Threshold technique yields a 71% reduction in solution time relative to AETG-SAT. We believe that additional improvements are possible by more tightly integrating History and Threshold. The current combination attempts to clearly separate the portions of the row in which each technique is active, however, it is possible to accelerate progress toward the threshold by counting the must assignments produced by History as making progress through the row. This integration would further reduce solution time without impacting solution quality – since Threshold will not overwrite must assignments from History.

These data demonstrate unequivocally that integrating constraints into CCIT solution algorithms can not only be efficient, but can actually make solution times significantly faster. This result may seem counter-intuitive at first, but it can be attributed to the fact that the techniques leverage SAT to aggressively prune the AETG search space. The cost benefits of this pruning more than compensate for the additional overhead of mining SAT solver data structures, maintaining threshold counters, and identifying implicit constraints during initialization.

4.6.6 Further Analysis of the Threshold

We performed additional analyses to examine the impact of increasing the covering array strength and of changing the characterization of the covering array on the threshold. First we examine the threshold when our algorithms are run for higher strength, i.e. different

values of t. When we run all of the same samples for $t = 3$ we see the threshold move to 80% (this threshold analysis is not shown). We also see a steady decrease in the resulting covering array size. This differs from the $t = 2$ data where our size was minimal at the 60% threshold. Our conjecture is that the random choices made by the SAT solver affect a much larger set when $t = 3$ which means we are less likely to make good choices by chance. Table 4.6 and Table 4.7 show size and time data correspondingly for our case study subjects when $t = 3$. Although our threshold is now 80% we still see a large time savings when we use AETG-Threshold. This is because the run times are much longer overall. For instance in gcc, the time saved by using an 80% threshold is approximately 8 hours. Our smallest arrays are found with AETG-History for $t = 3$. Which algorithmic technique provides the greatest cost savings will now depend on the cost of running tests for a single configuration versus the cost of constructing the covering array.

We also examined synthesized data sets for $t = 2$ that do not follow the characteristics of our case study subjects. Specifically, we decreased the number of factors, but increased the average number of values for each factor to be between 3-30 values. In this situation we also saw an increase in the threshold to 80%, leading us to conclude that the threshold will be sensitive to the parameters of the constrained covering array. Consequently, we believe that selection of specific threshold values is best determined by balancing the value of CMCA size versus generation time in a particular testing context.

4.6.7 Threats to Validity

The most significant threats to our findings is their generalization to other subjects. We cannot be sure that the subjects chosen and the respective simulated data are representative of all configurable software. We have, however, tried to control for this by using four different subjects from differing domains that have large user population bases. All

of these are open source programs, however, which may not be reflective of proprietary systems. We have also seen that the size data of our resulting samples, and the resulting threshold is sensitive to the parameters of the covering array. This means that both the software being modeled and the strength of the array may affect the threshold value. The time data, however, appears to be more stable across all experiments and should generalize better.

We have taken special measures to assure the internal validity of our findings. We have independently checked the constrained covering arrays through random sampling with different programs to confirm that they generate the correct constrained covering arrays. We have also validated the programs that are used to perform that checking.

While it is clear that one could develop many measures for judging the value of a CCIT method, we believe that CMCA size and generation time are the core elements of such an evaluation. Their relative weights in drawing value judgments may be varied depending on usage context, but we leave such exploration to future work.

4.7 Summary of the Work

The conventional wisdom in the CIT community is that constraints significantly complicate the problem of computing a CIT sample. In this chapter, we present a set of algorithms that synergistically integrate boolean satisfiability algorithms with greedy CIT generation algorithms. The most-efficient of these algorithms allows high-quality CIT samples to be computed in less than one-third the time of widely-used unconstrained CIT algorithms. Moreover, this performance benefit was observed on a collection of constrained CIT problems that reflect the richness of constraints found in real-world systems. We believe this represents a promising step in advancing CIT methods towards even broader applicability for the testing of highly-configurable software systems.

The key insight in this work is to leverage the fact that both CIT generation and SAT solver algorithms perform a search of the same space. By formulating constrained CIT sample generation as alternating phases of CIT and SAT search we leverage information from one search to inform the other. This leads to significant pruning of the CIT search, and reductions in execution time, while retaining the portions of the search space that contain high-quality solutions.

We believe that the techniques in this dissertation open the way for more aggressive scaling of the application of constrained CIT methods. In addition to scaling the size of subjects, an additional, and orthogonal, dimension of scaling is to consider higher CIT strength. While our evaluation considered mostly pair-wise CCIT it is likely that for mission-critical systems, engineers will target higher-order coverage which will dramatically increase the cost of CIT. The analyses we performed on some instances of CCIT for t=3 indicate that run times are dramatically larger than for t=2. We also observed that the threshold point for retaining high quality solutions increases. We have not yet explored the impact on strengths of $t > 3$. More study is needed to better understand the scalability of our CCIT methods to extremely large-scale highly-configurable mission-critical systems.

Covering Array, t=2			Time in Secs				
	camodel	**no. cons .**	**maetg**	**aetg sat**	**aetg hist**	**aetg thres 60%**	**aetg h-t 60%**
spins	$2^{13}4^{5}$	2^{13}	0.3	0.4	0.3	0.2	**0.2**
spinv	$2^{42}3^{2}4^{11}$	$2^{47}3^{2}$	8.2	11.3	8.5	4.5	**3.5**
gcc	$2^{189}3^{10}$	$2^{37}3^{3}$	221.7	286.9	204	70.2	**68.0**
apache	$2^{158}3^{8}4^{4}5^{1}6^{1}$	$2^{3}3^{1}4^{2}5^{1}$	258.3	249.2	244.1	**76.4**	76.5
bugz.	$2^{49}3^{1}4^{2}$	$2^{4}3^{1}$	4.5	6.2	4.4	1.9	**1.5**
1.	$2^{86}3^{3}4^{1}5^{5}6^{2}$	$2^{20}3^{3}4^{1}$	79.1	71.3	66.1	24.4	**22.2**
2.	$2^{86}3^{3}4^{3}5^{1}6^{1}$	$2^{19}3^{3}$	39.9	47	40.7	16.2	**14.7**
3.	$2^{27}4^{2}$	$2^{9}3^{1}$	0.9	0.9	0.8	0.4	**0.3**
4.	$2^{51}3^{4}4^{2}5^{1}$	$2^{15}3^{2}$	8.8	8.8	8.1	3.4	**3.1**
5.	$2^{155}3^{7}4^{3}5^{5}6^{4}$	$2^{32}3^{6}4^{1}$	404.3	404.8	372.9	134.8	**120.3**
6.	$2^{73}4^{3}6^{1}$	$2^{26}3^{4}$	21.0	25.1	22.1	8.2	**7.2**
7.	$2^{29}3^{1}$	$2^{13}3^{2}$	0.6	0.6	0.6	**0.3**	**0.3**
8.	$2^{109}3^{2}4^{2}5^{3}6^{3}$	$2^{32}3^{4}4^{1}$	117	131.8	148.3	45.1	**40.6**
9.	$2^{57}3^{1}4^{1}5^{1}6^{1}$	$2^{30}3^{7}$	10.6	9.4	8.2	3.5	**3.0**
10.	$2^{130}3^{6}4^{5}5^{2}6^{4}$	$2^{40}3^{7}$	257	261.5	230.5	82.8	**74.3**
11.	$2^{84}3^{4}4^{2}5^{2}6^{4}$	$2^{28}3^{4}$	67.7	80.2	68.9	26.8	**23.8**
12.	$2^{136}3^{4}4^{3}5^{1}6^{3}$	$2^{23}3^{4}$	235.0	228.3	212.8	75.6	**68.0**
13.	$2^{124}3^{4}4^{1}5^{2}6^{2}$	$2^{22}3^{4}$	145.2	153.2	146.8	60.1	**45.5**
14.	$2^{81}3^{5}4^{3}6^{3}$	$2^{13}3^{2}$	54.3	61.4	57.7	20.6	**20.2**
15.	$2^{50}3^{4}4^{1}5^{2}6^{1}$	$2^{20}3^{2}$	12.3	13.0	11.7	4.9	**4.7**
16.	$2^{81}3^{3}4^{2}6^{1}$	$2^{30}3^{4}$	26.9	35.1	27.7	11.1	**9.7**
17.	$2^{128}3^{3}4^{2}5^{1}6^{3}$	$2^{25}3^{4}$	169.6	173.1	156.3	52.0	**46.9**
18.	$2^{127}3^{2}4^{4}5^{6}6^{2}$	$2^{23}3^{4}4^{1}$	198.3	199.2	193.1	67.3	**60.1**
19.	$2^{172}3^{9}4^{9}5^{3}6^{4}$	$2^{38}3^{5}$	610.7	586	534.4	206.6	**168.4**
20.	$2^{138}3^{4}4^{5}5^{4}6^{7}$	$2^{42}3^{6}$	347.9	362.9	325.7	121.1	**109.9**
21.	$2^{76}3^{3}4^{2}5^{1}6^{3}$	$2^{40}3^{6}$	42.1	49.6	41.6	18.5	**14.3**
22.	$2^{73}3^{3}4^{3}$	$2^{31}3^{4}$	17.3	19.9	17.6	6.7	**5.9**
23.	$2^{25}3^{1}6^{1}$	$2^{13}3^{2}$	0.8	0.7	0.6	**0.3**	**0.3**
24.	$2^{110}3^{2}5^{3}6^{4}$	$2^{25}3^{4}$	124.2	134.7	128.9	46.4	**40.6**
25.	$2^{118}3^{6}4^{2}5^{2}6^{6}$	$2^{23}3^{3}4^{1}$	193.6	196.7	180.5	**63.4**	61.7
26.	$2^{87}3^{1}4^{3}5^{4}$	$2^{28}3^{4}$	45.5	50.6	47.5	20.7	**15.5**
27.	$2^{55}3^{2}4^{2}5^{1}6^{2}$	$2^{17}3^{3}$	15.7	20.0	15.1	5.4	**4.6**
28.	$2^{167}3^{16}4^{2}5^{3}6^{6}$	$2^{31}3^{6}$	584.5	588.5	546.3	180.9	**170.4**
29.	$2^{134}3^{7}5^{3}$	$2^{19}3^{3}$	148.1	142.7	135.9	51.7	**38.9**
30.	$2^{72}3^{4}4^{1}6^{2}$	$2^{20}3^{2}$	29.8	35.8	31.1	11.8	**10.3**
sum			4501.3	4646.6	4249.2	1524.3	**1356.3**

Table 4.4: Average Time Over 50 Runs

Covering Array, t=2			Size				
	camodel	**no. cons .**	**maetg size sat**	**aetg sat**	**aetg hist**	**aetg thres 60%**	**aetg h-t 60%**
spins	$2^{13}4^{5}$	2^{13}	26.3/8	27.1	27.1	**27.0**	27.1
spinv	$2^{42}3^{2}4^{11}$	$2^{47}3^{2}$	36.0/0	**42.5**	42.7	45.0	44.9
gcc	$2^{189}3^{10}$	$2^{37}3^{3}$	25.2/1	24.8	**24.7**	26.3	26.3
apache	$2^{158}3^{8}4^{4}5^{1}6^{1}$	$2^{3}3^{1}4^{2}5^{1}$	42.4/14	42.8	42.5	42.6	42.9
bugz.	$2^{49}3^{1}4^{2}$	$2^{4}3^{1}$	25.0/7	24.9	25.2	**21.8**	22.2
1.	$2^{86}3^{3}4^{1}5^{5}6^{2}$	$2^{20}3^{3}4^{1}$	56.5/0	54.7	55.4	**53.3**	53.7
2.	$2^{86}3^{3}4^{3}5^{1}6^{1}$	$2^{19}3^{3}$	40.0/0	40.1	**39.7**	40.4	40.5
3.	$2^{27}4^{2}$	$2^{9}3^{1}$	23.1/0	21.0	21.2	**20.7**	20.8
4.	$2^{51}3^{4}4^{2}5^{1}$	$2^{15}3^{2}$	29.8/1	28.7	28.7	28.9	**28.6**
5.	$2^{155}3^{7}4^{3}5^{5}6^{4}$	$2^{32}3^{6}4^{1}$	65.8/0	64.1	64.5	**63.8**	64.3
6.	$2^{73}4^{3}6^{1}$	$2^{26}3^{4}$	34.3/0	34.0	**33.9**	34.0	34.0
7.	$2^{29}3^{1}$	$2^{13}3^{2}$	12.4/0	12.0	**12.1**	12.5	12.5
8.	$2^{109}3^{2}4^{2}5^{3}6^{3}$	$2^{32}3^{4}4^{1}$	59.4/0	57.3	57.1	**55.6**	56.1
9.	$2^{57}3^{1}4^{1}5^{1}6^{1}$	$2^{30}3^{7}$	36.3/0	27.2	27.3	26.1	**26.0**
10.	$2^{130}3^{6}4^{5}5^{2}6^{4}$	$2^{40}3^{7}$	64.0/0	64.1	63.9	**60.3**	60.4
11.	$2^{84}3^{4}4^{2}5^{2}6^{4}$	$2^{28}3^{4}$	61.5/0	61.1	60.6	59.0	**58.3**
12.	$2^{136}3^{4}4^{3}5^{1}6^{3}$	$2^{23}3^{4}$	58.8/0	57.1	56.8	**54.3**	54.5
13.	$2^{124}3^{4}4^{1}5^{2}6^{2}$	$2^{22}3^{4}$	51.5/0	51.0	51.8	49.3	**48.6**
14.	$2^{81}3^{5}4^{3}6^{3}$	$2^{13}3^{2}$	56.6/3	56.2	56.0	**51.7**	51.8
15.	$2^{50}3^{4}4^{1}5^{2}6^{1}$	$2^{20}3^{2}$	41.1/1	40.7	40.7	**40.4**	40.7
16.	$2^{81}3^{3}4^{2}6^{1}$	$2^{30}3^{4}$	33.4/0	**33.0**	33.1	33.1	33.4
17.	$2^{128}3^{3}4^{2}5^{1}6^{3}$	$2^{25}3^{4}$	57.1/0	57.0	56.6	53.6	**53.4**
18.	$2^{127}3^{2}4^{4}5^{6}6^{2}$	$2^{23}3^{4}4^{1}$	59.5/0	57.7	58.3	**57.2**	57.3
19.	$2^{172}3^{9}4^{9}5^{3}6^{4}$	$2^{38}3^{5}$	68.0/0	66.3	65.7	**64.4**	64.7
20.	$2^{138}3^{4}4^{5}5^{4}6^{7}$	$2^{42}3^{6}$	72.6/0	71.6	71.8	**71.5**	71.8
21.	$2^{76}3^{3}4^{2}5^{1}6^{3}$	$2^{40}3^{6}$	56.0/0	55.0	54.7	**51.3**	51.7
22.	$2^{73}3^{3}4^{3}$	$2^{31}3^{4}$	28.8/0	28.7	28.8	**26.2**	26.4
23.	$2^{25}3^{1}6^{1}$	$2^{13}3^{2}$	20.7/1	**15.7**	**15.7**	16.3	16.0
24.	$2^{110}3^{2}5^{3}6^{4}$	$2^{25}3^{4}$	61.0/0	59.9	60.0	**58.1**	58.3
25.	$2^{118}3^{6}4^{2}5^{2}6^{6}$	$2^{23}3^{3}4^{1}$	67.6/0	66.2	66.4	**65.7**	**65.7**
26.	$2^{87}3^{1}4^{3}5^{4}$	$2^{28}3^{4}$	44.2/0	43.3	43.5	**42.0**	**42.0**
27.	$2^{55}3^{2}4^{2}5^{1}6^{2}$	$2^{17}3^{3}$	49.8/0	49.8	50.5	46.6	**45.8**
28.	$2^{167}3^{16}4^{2}5^{3}6^{6}$	$2^{31}3^{6}$	70.1/0	68.4	68.7	**68.4**	68.5
29.	$2^{134}3^{7}5^{3}$	$2^{19}3^{3}$	42.2/0	41.2	41.4	38.6	**38.4**
30.	$2^{72}3^{4}4^{1}6^{2}$	$2^{20}3^{2}$	49.2/0	50.2	50.4	**45.6**	45.8
sum			1626.4/36	1595.4	1597.7	**1551.6**	1553.3

Table 4.5: Average Size Over 50 Runs

Covering Array, t=3			Size				
	camodel	**no. cons.**	**maetg**	**aetg sat**	**aetg hist**	**aetg thres 80%**	**aetg h-t 80%**
spins	$2^{13}4^{5}$	2^{13}	105.9	117.5	117.1	118.1	118.4
spinv	$2^{42}3^{2}4^{11}$	$2^{47}3^{2}$	168.3	243.0	242.2	254.1	253.7
gcc	$2^{189}3^{10}$	$2^{37}3^{3}$	88.5	106.5	106.9	111.8	112.3
apache	$2^{158}3^{8}4^{4}5^{1}6^{1}$	$2^{3}3^{1}4^{2}5^{1}$	207.4	206.9	206.8	212.3	211.7
bugz.	$2^{49}3^{1}4^{2}$	$2^{4}3^{1}$	71.1	71.7	72.2	74.8	75.3
sum			641.3	745.7	745.2	771.2	771.4

Table 4.6: Average Size over 50 Runs for $t = 3$

Covering Array, t=3			Time in Secs				
	camodel	**no. cons.**	**maetg**	**aetg sat**	**aetg hist**	**aetg thres 80%**	**aetg h-t 80%**
spins	$2^{13}4^{5}$	2^{13}	5.9	7.7	6.2	4.9	4.1
spinv	$2^{42}3^{2}4^{11}$	$2^{47}3^{2}$	611.2	879.4	636.9	490.4	394.8
gcc	$2^{189}3^{10}$	$2^{37}3^{3}$	44498.2	54307.2	53226.8	28082.1	27102.1
apache	$2^{158}3^{8}4^{4}5^{1}6^{1}$	$2^{3}3^{1}4^{2}5^{1}$	60126.4	60216.3	60785.2	31210.1	30066.0
bugz.	$2^{49}3^{1}4^{2}$	$2^{4}3^{1}$	199.4	209.6	190.1	123.9	103.8
sum			105441.1	115620.2	114845.2	59911.3	57670.7

Table 4.7: Average Time over 50 Runs for $t = 3$

Chapter 5

Integration Testing of Software Product Lines Using Compositional Symbolic Execution[1]

So far we have tackled the large product space of an SPL from the sampling perspective. More specifically, we developed an interaction coverage and a method to generate a sample to fulfill such coverage. Next we will tackle the same challenge from the reuse perspective. More specifically, we want to integration test all interactions of an SPL by reusing testing results of smaller-size interactions for testing larger-size interactions.

As discussed in 1.2.3, we exploit inclusion relations among interactions to reuse lower-level interaction test results for testing higher-level interactions, so that our testing method is more efficient. There are several steps to reach this reuse goal, and we show an overview next.

[1]Portions of the material presented in this chapter have appeared previously in [122]. The material presented in this chapter provides a running example for the compositional symbolic execution algorithm.

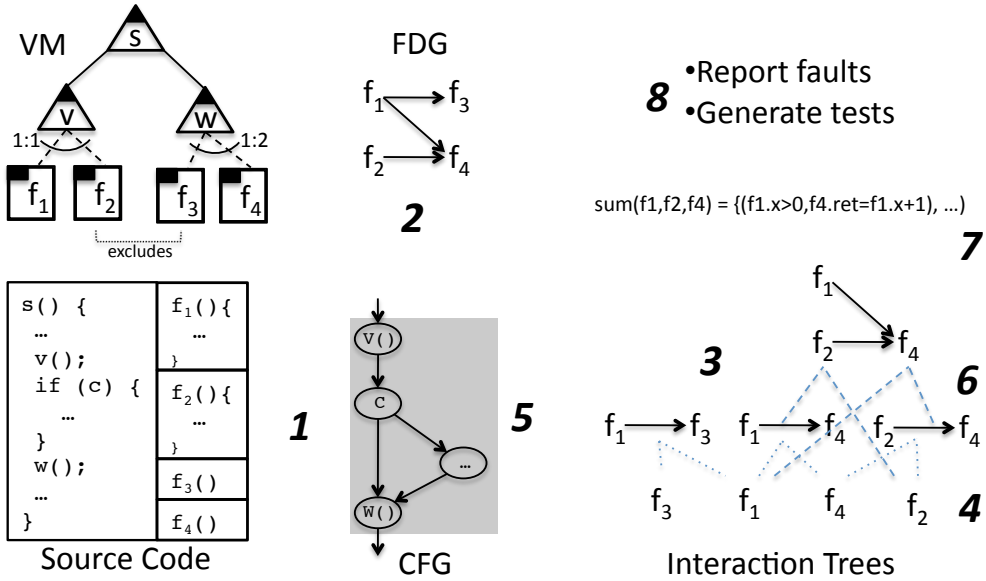


Figure 5.1: Conceptual Overview of Compositional SPL Analysis

5.1 Overview – Dependence driven Compositional Analysis

Our technique exploits an SPL's variability model and the inter-dependence of feature implementations to reduce the cost of applying symbolic execution to reason about feature interactions. Figure 5.1 provides a conceptual overview.

An SPL is comprised of a source code base and an OVM. The OVM and its constraints (e.g., the excludes between f_2 and f_3) defines the set of features that may be present in an instance of the SPL.

Our technique begins (step are denoted by large bold italic numerals in the figure) by applying standard control flow and dependence analyses on the code base. The former results in a control flow graph (CFG) and the latter results in a program dependence graph (PDG). In step 2, the PDG is analyzed to calculate a feature dependence graph (FDG) which reflects inter-feature dependences. The edges of the FDG are pruned to be

consistent with the OVM, e.g., the edge from f_2 to f_3 is not present.

Step 3 involves the calculation, from the FDG, of the hierarchy of all k-way feature interaction trees. The structure of this hierarchy reflects how lower-order interactions can be composed to create higher-order interactions. For instance, how the interaction among f_1, f_2, and f_4 can be constructed by combining f_1 with an existing interaction for f_2 and f_4.

The interaction tree hierarchy is used to guide the calculation of symbolic summaries for all interaction trees in a compositional fashion. This begins, in Step 4, by applying symbolic execution to the source code of the individual features in isolation. When composing two existing summaries, for example f_1 and f_3, to create a 2-way interaction tree, a summary of the behavior of the common SPL code which leads between those summaries must be calculated. Step 5 achieves this by locating the calls to the features in the CFG and calculating a chop [115] – shown as the shaded figure in the CFG – the edges of the chop are used to guide a customized symbolic execution to produce an edge summary. In step 6, a pair of existing lower-order interaction summaries and the edge summary are composed to produce a higher-order summary – such a summary is illustrated at point 7 in the figure.

In step 8, summaries can be exploited to detect faults, via comparison to fault oracles, or to generate tests by solving the constraints generated by symbolic execution and composition. We describe the major elements next.

5.2 Relating SPL Models To Implementations

An SPL implementation can be partitioned into *regions* of code that implement each feature; the remaining code implements the common functionality shared by all SPL instances. There are many implementation mechanisms for realizing variability in a code

base [71]. Our methodology can target these by adapting the summary computation for Step 4 and feature dependence graph construction for Step 2, but for simplicity it suffices to view features as methods where common code makes calls on those methods.

In the remainder of this section, we assume the existence of a mapping from in the OVM to methods in a code base; we use the name of a feature to denote the method when no confusion will arise. Features can be called from multiple points in the common code, but to simplify the presentation of our technique, we assume each feature is called from a single call site.

Given a pair of features, f_1 and f_2, where the call to f_2 is reachable in the CFG from the call to f_1, their *common region* is the source code chop [115] arising when the calls are used as the chop criterion. This chop is a single-entry single-exit sub-graph of the program control flow graph (CFG) where the entry node is the call to f_1 and the exit node is the call to f_2. The CFG paths within the chop overapproximate the set of feasible program executions that can lead from the return of f_1 to the call to f_2. These chops play an important role in accounting for the composite behavior of features as mediated by common code.

5.3 Calculating Feature Interactions

We leverage the concept of program dependences, and the PDG [105], to determine inter-feature dependences. A PDG is a directed graph, (S, E_{PDG}), whose vertices are program statements, S, and $(s_i, s_j) \in E_{PDG}$ if s_i defines the value of a location that is subsequently read at s_j. A feature dependence graph (FDG) is an abstraction of the PDG for an SPL implementation.

Definition 5.3.1 (Feature Dependence Graph) *Given a PDG for an SPL, (S, E_{PDG}), the FDG, (F, E_{FDG}), is a directed graph whose vertices are features, F, and $(f_i, f_j) \in E_{FDG}$ iff*

$\exists s_i, s_j \in S : s_i \in S(f_i) \wedge s_j \in S(f_j) \wedge (s_i, s_j) \in E_{PDG}$ *where* $S(f)$ *is the set of statements in feature* f.

We capture the interaction among features by defining a tree that is embedded in the FDG. The intuition is that the root is the sink of a set of feature dependence edges. The output values of that root feature reflect the final interaction effects, and are defined in terms of the input values of the features that form the leaves of the tree.

Definition 5.3.2 (Interaction Tree) *Given an FDG,* (F, E_{FDG})*, a k-way interaction tree is an acyclic, connected, simple subgraph,* (F', E')*, where* $F' \subseteq F$*,* $E' \subseteq E_{FDG}$*,* $|F'| = k$*, and where* $\exists r \in F' : \forall v \in F' : r \in v.(E')^*$*. We call the common reachable vertex the root of the interaction tree.*

The set of all k-way interaction trees for an SPL can be constructed as shown in Algorithm 6. The algorithm uses a constructor $tree()$ which, optionally, takes an existing tree and adds edges to it expanding the set of vertices as appropriate. For a tree, t, the set of vertices is $v(t)$ and the root is $root(t)$. Before adding a tree, the set of features in the tree must be checked to ensure they are consistent with the OVM; this is done using the predicate $consistent()$.

The algorithm accepts k and an FDG and returns the set of k-way interactions. It builds the set of interactions incrementally. For an i-way interaction, it extends an $i-1$-way interaction by adding a single additional vertex and an edge. While other strategies for building interaction trees are possible, this approach has the advantage of efficiency and simplicity. Based on our case studies, reported in Section 5.5, this approach is sufficient to enable significant improvement over more standard analyses of an SPL code base.

Interaction trees can be organized hierarchically based on their structure.

Definition 5.3.3 (Interaction Hierarchy) *Given a k-way interaction tree,* $t_k = (F, E)$*, where*

```
1:  interactionTrees(k, (F, E))
2:    T := ∅
3:    for (f_i, f_j) ∈ E
4:      T ∪ = tree(f_i, f_j)
5:    for i = 3 to k + 1
6:      for t_{i-1} ∈ T ∧ |t_{i-1}| = i − 1
7:        for v ∈ F − v(t_{i-1})
8:          if (root(t_{i-1}), v) ∈ E ∧ consistent(v(t_{i-1} ∪ v)) then
9:            T ∪ = tree(t_{i-1}, (root(t_{i-1}), v))
10:         else
11:           for (v, v') ∈ E ∧ v' ∈ v(t_{i-1})
12:             if consistent(v(t_{i-1} ∪ v) then T ∪ = tree(t_{i-1}, (v, v'))
13:         endif
14:   return T
15: end interactionTrees()
```

Algorithm 6: Computing k-way Interaction Trees

$k > 1$, we can define a pair of interaction trees $t_i = (F_i, E_i)$ and $t_j = (F_j, E_j)$, such that $F_i \cap F_j = \emptyset$, $|F_i| + |F_j| = k$, and $\exists (f_i, f_j) \in E$. We say that t_k is the parent of t_i and t_j and, conversely, that t_i and t_j are the children of t_k.

The base case of the hierarchy, where $k = 1$, is simply each feature in isolation. There are many ways to construct such an interaction hierarchy, since for any given k-way interaction tree cutting a single edge partitions the tree into two children. As discussed below, the hierarchy resulting from Algorithm 6 enjoys a structure that can be exploited in generating summaries of interaction pattern behavior. The parent (child) relationships among interaction trees can be recorded at the point where the *tree*() constructor calls are made in Algorithm 6.

Figure 5.2 shows a simple FDG with 4 features and 3 directed interactions. Here one edge represents a def-use relation underlying two features. There are three tables to show all interactions from 2- to 4-way. Each table includes both normal interactions and directed interactions. For example, for a 3-way interaction (f2, f3, f4), we have a rooted tree which includes exactly 2 edges ($f2 \rightarrow f3$ and $f3 \rightarrow f4$) connecting 3 nodes (f2, f3 and f4) with one sink (f4).

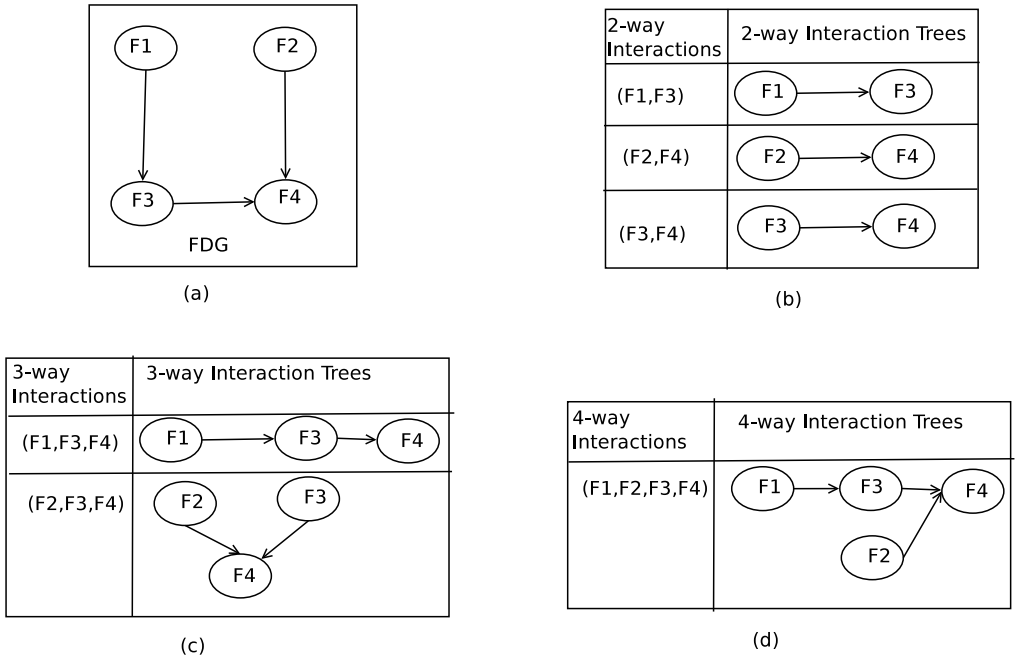


Figure 5.2: Traditional Interactions and Interaction Trees

5.4 Composing Feature Summaries

Our goal is to analyze program paths that span sets of features in an SPL to support fault detection and test generation. Our approach to feature summarization involves two distinct phases: (1) the application of bounded symbolic execution to feature implementations in isolation to produce feature summaries, and (2) the matching and combination of feature summaries to produce summaries of the behavior of interaction patterns.

Phase (1) is performed by applying traditional symbolic execution where the length of the longest branch sequence is bounded to d – the depth. For each feature, f, this results in a summary, f_{sum}, as defined in Section 2.4.4.

When performing symbolic execution of f there are three possible outcomes: (a) a complete execution of f which returns normally as analyzed within d branches, (b) an exception, including assertion violations, is detected before d branches are explored, and (c) the depth bound is reached. In our work, we only accumulate the outcomes falling

into (a) into f_{sum}.

Case (b) is interesting, because it *may* indicate a fault in feature f. The isolated symbolic execution of f allows for any possible state on entry to the feature, however, it is possible that a detected exception is infeasible in the context of a system execution. In future work, we will preserving results from case (b) and attempt to determine their feasibility when composed in interaction patterns with other features – this would reduce and, when interaction patterns are sufficiently large, eliminate false reports of exceptions.

For phase (2) we exploit the structure of the interaction hierarchy resulting from the application of Algorithm 6 to generate a summary for a k-way interaction. As discussed above, such an interaction has (potentially several) pairs of children. It suffices to select any of those pairs.

Within each pair there is a $k-1$-way interaction, i, which we assume has a summary $i_{sum} = (pc_i, w_i)$, and single feature, f, summarized as $f_{sum} = (pc_f, w_f)$, which is connected by a single edge connected to either $root(i)$ or one of i's leaves, l. To compose i_{sum} and f_{sum} we must characterize the behavior of the FDG edge.

The existence of an edge (f, f') means that there is a common region beginning at the return from f and ending at the call to f'. Calculating the chop that circumscribes the CFG for this region allows us to label branch outcomes that lie within the chop and to direct the symbolic execution along paths from f that reach f'.

Algorithm 7 defines this approach to calculating edge summaries. It consists of a customized depth-bounded symbolic execution that only explores a branch if that branch lies within the chop for the common region. The algorithm makes use of several helper functions. Functions determine whether an instruction is a branch, $branch()$, the target of a branch, $target()$, and the symbolic expression for a branch given a symbolic state, $cond()$. Functions to calculate the successor of an instruction, $succ()$, the set of locations written by an instruction, $write()$, and updating the symbolic state based on an instruction,

```
1: eSum(E, l, e, pc, s, w, d)
2: if |pc| > 0
3:   if branch(l)
4:     l_t := target(l, true)
5:     if SAT(cond(l, s)) ∧ (l, l_t) ∈ E
6:       eSum(E, l_t, e, pc ∧ cond(l, s), s, w, d − 1)
7:     l_f := target(l, false)
8:     if SAT(¬cond(l, s)) ∧ (l, l_f) ∈ E
9:       eSum(E, l_f, e, pc ∧ ¬cond(l, s), s, w, d − 1)
10:  else
11:    if l = e
12:      sum ∪ = (pc, π(s, w))
13:    else
14:      s := update(s, l)
15:      w ∪ = write(l)
16:      eSum(E, succ(l), e, pc, s, w, d)
17:    endif
18: endif
19: if pc = true return sum
20: end eSum()
```

Algorithm 7: Edge Summary

$update()$, are also used. The $SAT()$ predicate determines whether a logical formula is satisfiable. Finally, the $\pi()$ function projects a symbolic state onto a set of locations.

$eSum(E_{chop}, succ(f), f', true, \emptyset, \emptyset, d)$ returns the symbolic summary for edge (f, f') where the parameters are as follows. E_{chop} is the set of edges in the CFG chop bounded by the return of f and the call to f', $succ(f)$ is the location at which initiate symbolic execution and f' is the call that terminates symbolic execution. $true$ is the initial path condition. The next two parameters are the initial symbolic state and the set of locations written on the path – both are initially empty. d is the bound on the length of the path condition that will be explored in producing the summary.

To produce a symbolic summary for the k-way interaction, we now compose i_{sum}, f_{sum}, and the edge summary computed by $eSum()$. There are two cases to consider. If the feature, f', is connected to $root(i)$ with an edge, $(root(i), f')$ we compose summaries in the

```
1:  cSum(s, s')
2:  s_c := ∅
3:  for (pc, w) ∈ s
4:    for (pc', w') ∈ s'
5:      eq := true
6:      for l ∈ read(pc')
7:        if ∃l ∈ dom(w)
8:          eq := eq ∧ input(s', l) = w(l)
9:      if SAT(pc ∧ eq ∧ pc')
10:       for l ∈ dom(w')
11:         if ∃l ∈ dom(w)
12:           w := w − (l, _)
13:       endfor
14:       s_c ∪ = (pc ∧ eq ∧ pc', w ∧ w')
15:     endif
16:   endfor
17: end cSum()
```

Algorithm 8: Composing Summaries

following order: $i_{sum}, (root(i), f')_{sum}, f'_{sum}$. If the feature, f', is connected to a leaf of i, l_i, with an edge, (f', l_i) we compose summaries in the following order: $f'_{sum}, (f', l_i)_{sum}, i_{sum}$.

Order matters in composing summaries because the set of written locations of two summaries may overlap and simply conjoining the equality constraints on the values at such locations will likely result in constraints that are unsatisfiable. We keep only last write of locations in a composed summary to honor the sequencing of writes and reads of locations that arise due to the order of composition.

Consider the composition of summary s with summary s', in that order. Let $(pc, w) \in s$ and $(pc', w') \in s'$ be two elements of those summaries. The concern is that $dom(w) \cap dom(w') \neq \emptyset$, where $dom()$ extracts the set of locations used to index into a map. Our goal is to eliminate the constraints in w on locations in $dom(w) \cap dom(w')$. In general, pc' will read the value of at least one location, l, and that location may have been written by the preceding summary. In such a case, the input value referenced in pc' should be equated to $w(l)$. Algorithm 8 composes two summaries taking care of these two issues.

In our approach, the generation of a symbolic summary produces "fresh" symbolic variables to name the values of inputs. A map, $input()$, records the relationship between input locations and those variables. We write $input(s, l)$ to denote a summary s and a location l to access the symbolic variable. For a given path condition, pc, a call to $read(pc)$ returns the set of locations referenced in the constraint – it does this by mapping back from symbolic variables to the associated input locations. We rely on these utility functions in Algorithm 8.

Algorithm 8 considers all pairs of summary elements and generates, through the analysis of the locations that are written by the first summary and read by the second summary, a set of equality constraints that encode the path condition of the second summary element in terms of the inputs of the first. The pair of path conditions along with these equality constraints are checked for satisfiability. If they are satisfiable, then the cumulative write effects of the summary composition are constructed. All of the writes of the later summary are enforced and the writes in the first that are shadowed by the second are eliminated – which eliminates the possibility of false inconsistency.

5.4.1 Complexity and Optimizion of Summary Composition

From studying the Algorithm 8 it is apparent that the worst-case cost of constructing all summaries up to k-way summaries is exponential in k. This is due to the quadratic nature of the composition algorithm.

In practice we see quite a different story, in large part because we have optimized summary composition significantly. First, when we can determine that a pair of elements from a summary that might potentially match we ensure that for any shared features the summaries agree on the values for the elements of those summaries; this can be achieved through a string comparison of the summary constraints which is much less

expensive than calling the SAT solver. Second, we can efficiently scan for constraints in one summary that are not involved in another summary and those can be eliminated since they were already found to be satisfiable in previous summary analyses.

5.4.2 Composing Summaries Example

We illustrate the above steps with the bank SPL in Figure 1.1 which has its interaction tree hierarchy in Figure 1.4. We further assume that Table 5.1 shows summaries of single features in the bank SPL.

Recall that a summary of a single feature is a set of pair of partition and effects. For example, Report in Table 5.1 has two pairs which are (B>0, return2=B) and (B≤0, return2=B-2). Then the first step for composing summary is to find the connection information between A and B. The connection information is a series of equal clauses which connects escaped variables of A with inputs variables of B.

After we get summaries of A and B and connection information from A to B, we then chain each summary of A with each summary of B with a boolean conjunctive operator considering the connection information, and finally we check if such a chained formula is satisfiable based on a decision procedure. For example, Table 5.2 shows all composed summaries for all three 2-way directed interactions in Figure 1.4. For $Tya \rightarrow Smy$, there are 4 possible summary combinations, but only 3 are satisfiable. For the first one composed summary, we can see there are 3 parts separated with 3 pairs of parentheses. The first part is the second summary from Tya. The second part is the connection information which connects a return variable, $return3$, with an input variable, A. Finally, the third part is the second summary from Smy. We can see an assignment model $(C = 0)$ will satisfy the whole new chained formula, $(C = 0 \wedge return3 = C - 3) \wedge (return3 = A) \wedge (\neg(A > 0 \wedge F > 0) \wedge return1 = A + 1)$.

So far we have discussed steps to compose summaries from two single summaries, next we discuss how to compose summaries from two composed summaries. The key is to use conjunctive operators to chain each summary of one interaction with each summary of another. Table 5.3 includes four summaries of a 3-way directed interaction based on three summaries of two 2-way directed interactions. For the first summary, it consists of two parts, 1.1 and 1.2. They are chained together with an $\wedge$ operator. We separate them as two parts just for easy explanation. The first part is from the first summary of the 2-way directed interaction, Tya $\rightarrow$ Smy. The second part is from the second summary of another 2-way directed interaction, Pay$\rightarrow$Smy. There are a total of 9 possible summary combinations, but only these 4 are satisfiable.

Now we can see in the interaction tree hierarchy that both Tya$\rightarrow$Smy and Pay$\rightarrow$Smy are used to compose the directed interaction { Tya$\rightarrow$Smy,Pay$\rightarrow$Smy}. Correspondingly in the summary composition, we do not compute the summary of this 3-way directed interaction from the scratch; instead we reuse summaries of these 2-way interactions to compute the summary of the 3-way. This reflects our reuse mechanism during integration testing interactions by exploiting the similarity of an SPL.

Features	Summaries
Summary(Smy)	1. (A>0 $\wedge$ F>0, return1=A) 2.($\neg$(A>0 $\wedge$F>0), return1=A+1)
Report(Rep)	1. (B>0 , return2=B) 2. (B$\leq$0 , return2=B-2)
To_your_account (Tya)	1. (C$\neq$0 , return3=C) 2. (C=0 , return3=C-3)
Pay	1. (D>-5, return4=D) 2. (D$\leq$-5 , return4=D-5)
To_one_country(Toc)	1. (E$\neq$10 , return5=E) 2. (E=10 , return5=E-11)

Table 5.1: Summaries of Single Features

5.5 Case Study

We have designed a case study for evaluating the feasibility of our approach that ask the following two research questions. (**R**Q1): What is the reduction from our dependency

2-way Trees	Summaries
Tya $\rightarrow$ Smy	1. $(C = 0 \wedge return3 = C - 3) \wedge (return3 = A) \wedge (\neg(A > 0 \wedge F > 0) \wedge return1 = A + 1)$
	2. $(C \neq 0 \wedge return3 = C) \wedge (return3 = A) \wedge (A > 0 \wedge F > 0 \wedge return1 = A)$
	3. $(C \neq 0 \wedge return3 = C) \wedge (return3 = A) \wedge (\neg(A > 0 \wedge F > 0) \wedge return1 = A + 1)$
Pay $\rightarrow$ Smy	1. $(D > -5 \wedge return4 = D) \wedge (return4 = F) \wedge (A > 0 \wedge F > 0 \wedge return1 = A)$
	2. $(D > -5 \wedge return4 = D) \wedge (return4 = F) \wedge (\neg(A > 0 \wedge F > 0) \wedge return1 = A + 1)$
	3. $(D <= -5 \wedge return4 = D - 5) \wedge (return4 = F) \wedge (\neg(A > 0 \wedge F > 0) \wedge return1 = A + 1)$
Toc$\rightarrow$ Rep	1. $(E \neq 10 \wedge return5 = E) \wedge (return5 = B) \wedge (B > 0 \wedge return2 = B)$
	2. $(E \neq 10 \wedge return5 = E) \wedge (return5 = B) \wedge (B \leq 0 \wedge return2 = B - 2)$
	3. $(E = 10 \wedge return5 = E - 11) \wedge (return5 = B) \wedge (B \leq 0 \wedge return2 = B - 2)$

Table 5.2: Summaries of 2-way Directed Interactions

3-way Trees	Summaries
Tya $\rightarrow$ Smy	**1.1** $((C = 0 \wedge return3 = C - 3) \wedge (return3 = A) \wedge (\neg(A > 0 \wedge F > 0) \wedge return1 = A + 1)) \wedge$ **1.2** $((D > -5 \wedge return4 = D) \wedge (return4 = F) \wedge (\neg(A > 0 \wedge F > 0) \wedge return1 = A + 1))$
	2.1 $((C \neq 0 \wedge return3 = C) \wedge (return3 = A) \wedge (A > 0 \wedge F > 0 \wedge return1 = A)) \wedge$ **2.2** $((D > -5 \wedge return4 = D) \wedge (return4 = F) \wedge (A > 0 \wedge F > 0 \wedge return1 = A))$
Pay $\rightarrow$ Smy	**3.1** $((C \neq 0 \wedge return3 = C) \wedge (return3 = A) \wedge (\neg(A > 0 \wedge F > 0) \wedge return1 = A + 1)) \wedge$ **3.2** $((D > -5 \wedge return4 = D) \wedge (return4 = F) \wedge (\neg(A > 0 \wedge F > 0) \wedge return1 = A + 1))$
	4.1 $((C \neq 0 \wedge return3 = C) \wedge (return3 = A) \wedge (\neg(A > 0 \wedge F > 0) \wedge return1 = A + 1)) \wedge$ **4.2** $((D \leq -5 \wedge return4 = D - 5) \wedge (return4 = F) \wedge (\neg(A > 0 \wedge F > 0) \wedge return1 = A + 1))$

Table 5.3: Summaries of 3-way Directed Interactions

analysis on the number of interactions that should be tested in an SPL? (**R**Q2): What is the difference in time between using our compositional symbolic technique versus a traditional directed technique?

5.5.1 Objects of Analysis

We selected two software product lines. The first SPL is based on the implementation of the Software Communication Architecture-Reference Implementation (SCARI-Open v2.2) [30] and the second is a graph product line, GPL[87, 78] used in several other papers on SPL testing.

The first product line, SCARI, was constructed by us as follows. First we began with the Java implementation of the framework. We removed the non-essential part of the product line (e.g. logging, product installation and launching) and features that required

CORBA Libraries to execute. For instance, the CORBA framework builds a distributed computing environment for a generated product. In such an enviroment, modulation can be on one machine, and demodulation may be on another machine. For simplicity, we keep only data flow relations between modulation and demodulation, but we remove the code related to CORBA that distributes these to different machines. We kept the core mandatory feature, Audio Device, and transformed four features that were written in C (ModFM, DemodFM, Chorus and Echo), into Java. We then added 9 other features which we translated from C to Java from the GNU Open Source Radio [51] and the Sound Exchange (SoX), site [124]. Table 5.4 shows the origin of each feature and the number of summaries for each. We used the example function for assembling features, to write a configuration program that composes the features together into products. The feature model is shown in Figure 5.3(a).

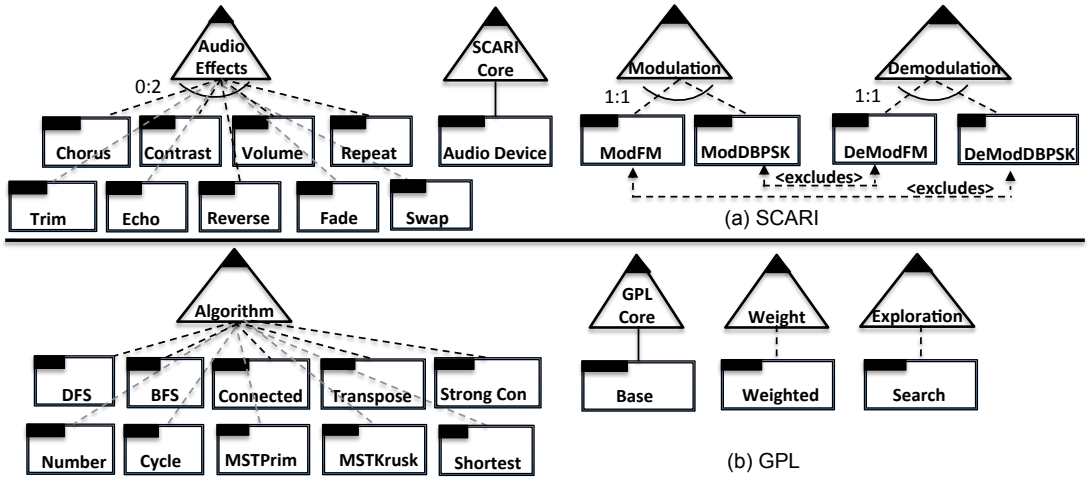


Figure 5.3: Feature Models for (a) SCARI and (b) GPL

The graph product line (GPL) [87] has been used for various studies on SPLs. We start with the version found in the implementation site for [78]. To fit our prototype tool, we refactored some code so that every feature is contained in a method. We removed several features because either we could not find a method in the source code or because JPF

Features	Origin	LOC	No. Summaries
Chorus	[30]	30	6
Contrast	[124]	14	5
Volume	[124]	47	5
Repeat	[124]	12	3
Trim	[124]	11	6
Echo	[30]	31	5
Reverse	[124]	14	4
Fade	[124]	9	4
Swap	[124]	27	4
AudioDevice	[30]	13	3
ModFM	[30]	19	4
ModDBPSK	[51]	6	2
DemodFM	[30]	18	4
DemodDBPSK	[51]	6	3
Total		257	58

Table 5.4: SCARI Size by Feature

Features	LOC	No. Summaries
Base	85	56
Weighted	32	148
Search	35	19
DFS	23	41
BFS	23	6
Connected	4	8
Transpose	27	3
StronglyConnected	19	9
Number	2	2
Cycle	40	19
MSTPrim	92	4
MSTKruskal	106	3
Shortest	102	3
Total	590	321

Table 5.5: GPL Size by Feature

would not run. For example, Base___ is a feature in their variability model represented as a context free grammar and in the corresponding configuration file, but there are no corresponding statements/methods in the source code. The feature, Benchmark, involves file read and write operations, which are not supported well in JPF. Though we believe that the remaining features of GPL still illustrate the feasibility of the compositional symbolic execution, we consider the use of industry strength symbolic executors and more subjects for validating our method as a part of future work. We made the method Prog our main entry point for the program. We did not include any constraints for simplicity. Figure 5.3(b) shows the resulting feature model and Table 5.5 shows the number of lines of code and the number of summaries by feature.

5.5.2 Method and Metrics

Experiments are run on an AMD Linux computing cluster running CentOS 5.3 with 128GB memory per node. We use Java Pathfinder (JPF) [100] to perform SE with the Choco solver for SCARI and CVC3BitVector for GPL. We adapt the information flow

analysis (IFA) package [58] in Soot [141] for our FDG. In SCARI we use the configuration program for a starting point of analysis. In GPL we use the Prog program, which is an under-approximation of the FDG.

For RQ1 we compute the number of possible interactions (directed and undirected) at increasing values for k, obtained directly from the feature model. We compare this with the number that we get from the interaction trees. For RQ2, we compare the time that is required to execute the two symbolic techniques on all of the trees for increasing values of k. We compare incremental SE (**I**ncComp) and a full direct SE (**D**irectSE). We set the depth for SE at 20 for IncComp and allow DirectSE k-times that depth since it works on the full partial-product each time, while IncComp composes k summaries each computed at depth 20. DirectSE does not use summaries, but in the SPLs we studied there is no opportunity for summary reuse within the analysis of a partial product – our technique reuses summaries across partial products.

5.5.3 Results

RQ1. Table 5.6 compares the number of interactions obtained from just the OVM with the number of interaction trees obtained through our dependency analysis. We present k from 2 to 5. The column labelled UI is the number of interactions calculated from all k-way combinations of features. In SCARI there are only three true points of variation given the model and constraints, therefore we see the same number of interactions for $k = 3$ and 4. For $k = 5$, we have fewer interactions since there are 5 unique 4-way feature combinations in a single product with 5 features, but only a single 5-way combination. The DI column represents the number of directed interactions or all permutations ($k! \times UI$). The next two columns are feasible interactions obtained from the interaction trees. Feasible UI, removes direction, counting all trees with the same features as equivalent. Feasible DI is

Subject	k	UI	DI	Feasible UI	Feasible DI	UI Reduction	DI Reduction
SCARI	2	188	376	85	85	54.8%	77.4%
	3	532	3192	92	92	82.7%	97.1%
	4	532	12768	162	162	69.5%	98.7%
	5	164	19680	144	144	12.2%	99.3%
GPL	2	288	576	21	27	92.7%	95.3%
	3	2024	12144	29	84	98.6%	99.3%
	4	9680	232320	31	260	99.7%	99.9%
	5	33264	3991680	20	525	99.9%	100.0%

Table 5.6: Reduction for Undirected (U) and Directed (D) Interactions (I)

the full tree count. The last two columns give the percent reduction. For the undirected interactions we see a reduction of between 12.2% and 99.9% across subjects and values of k, and the reduction is more dramatic in GPL (92.7%-99.9%). If we consider the directed interactions, which would be needed for test generation, there is a reduction ranging from 77.4% to 100%. In terms of absolute values we see a reduction in GPL from over 3 million directed interactions at $k = 5$, down to 525, an order 4 magnitude of difference. DIs are useful to detect more behaviors. For example, given a one-second-sound file, trim→repeat removes 1-second-sound and generates an empty file; repeat→trim repeats the sound once and outputs a 1-second-sound file. For SCARI, we also see a bigger difference between UI Reduction and DI Reduction than GPL. The root reason is because GPL has no constraints in the OVM model, which generates a larger interaction space than SCARI. At the same time, we use Prog as the main entry point to calculate FDG and trees, which may under-approximate the number of trees. Both facts lead to a big reduction for both UI Reduction and DI Reduction in GPL, which are close to 100%. As a result, this leads to a small difference between both reductions for GPL.

RQ2. Table 5.7 compares the performance of DirectSE and IncComp in terms of time (in seconds). It lists the number of directed (D) and undirected (U) interactions (I) for each k, that are feasible based on the interaction trees. Some features in the feature models may have more than one method. In RQ1 based on the OVM we reported interactions only at

Subject	k	Feasbile UI	Feasible DI	DirectSE	IncComp	
				Time (sec)	Time (sec)	SAT/SMT, Avoided Calls
SCARI	1	14	14	6.75	6.75	58
	2	85	85	14.48	9.63	430/1780, 0
	3	92	92	17.67	10.06	844/2226, 1587
	4	162	162	36.09	10.93	1505/2909, 3442
	5	144	144	35.87	11.70	2075/3523, 5696
GPL	1	49	49	41.77	41.77	321
	2	60	76	67.25	56.28	663/985, 0
	3	81	310	184.76	82.00	1441/1901, 1809
	4	82	1725	727.34	216.63	5814/7342, 5396
	5	52	8135	3887.23	965.92	27444/34147, 19743

Table 5.7: Time comparisions for SCARI and GPL

the feature level. However in this table, we consider all methods within a feature and give a more precise count of the interactions; we list all of the interactions (both directed and undirected) between features. The next two columns present time. For Direct SE we re-start the process for each k, but for the IncComp technique we use cumulative times because we must first complete $k-1$ to compute k. Although both techniques use the same time for single feature summaries, they begin to diverge quickly. DirectSE is 3 times slower for $k = 5$ on SCARI, and 4 times slower on GPL. Within SCARI we see no more than a 3 second increase to compute $k+1$ from k (compared to 14-35 seconds for DirectSE) and in GPL we see at most 750 (12 mins). For DirectSE it requires as long as 3160 (1̃ hour).

The last column of this table shows how many feasible paths were sent to the SAT solver (SAT). We saw (but don't report) a similar number for DirectSE which we attribute to our depth bounding heuristic. The number for SMT represents the total number of possible calls that were made to the SAT solver. However, we did not send all possible calls, because our matching heuristic culled out a number which we show as Avoided Calls.

5.6 Summary of the Work

In this chapter we have presented a compositional symbolic execution technique for integration testing of software product lines. Using interaction trees to guide incremental summary composition we can efficiently account for all possible interactions between features. We consider interactions as directed which gives us a more precise notion of interaction than previous research. In a feasibility study we have shown that we can (1) reduce the number of interactions to be tested by a factor of between 12.2 and 99.9% over an uninformed model, and (2) reduce the time taken to perform symbolic execution by as much as factor of 4 over a directed symbolic execution technique. Another advantage of this technique is that since our results and costs are cumulative, we can keep increasing k as time allows, making our testing stronger, without any extraneous work along the way.

As future work we plan to exploit the information gained from our analysis to perform directed test generation. By using the complete paths we can generate test cases from the constraints that can be used with more refined oracles. For paths which reach the depth bound, we plan to explore ways to characterize these partial paths to guide other forms of testing, such as random testing, to explore the behavior which is otherwise unknown.

Chapter 6

Conclusions and Future Work

In this chapter, we summarize challenges and solutions discussed in this dissertation, and then give three directions for future work.

6.1 Summary

SPLE is an increasingly important software engineering methodology for developing a set of similar products. In software testing, the biggest challenge is how we can guarantee the correctness for a huge number of products. Obviously we can not test every product to guarantee an error-free SPL. Although there have been many techniques presented in the literature, as discussed in Chapter 2, there is still much room for improvement in testing an SPL as a whole.

We proposed our methods from two perspectives: sampling and reuse. For sampling, there are two steps: special coverage criteria and sampling generation methods. For coverage criteria, we introduced criteria related to the constraints and interactions. For sampling generation methods, we extended current CIT techniques to generate a sample to fulfill the coverage with the consideration of constraints. More specifically, we proposed

one basic method with three optimizations. With our optimizations and the empirical study, we have solid evidence that we can generate a high-quality covering array in much less time for CCIT problems.

There are many different views to exploit the reuse concept for saving testing efforts. We took an integration testing perspective, which involves testing all directed interactions from 2- to k-way in an SPL. We organized these directed interactions from the bottom up as an interaction tree hierarchy, and then composed summaries of higher-level interaction trees by reusing summaries of lower-level interaction trees. From the preliminary experiment, we have two conclusions: 1) compositional symbolic execution is faster than direct symbolic execution; and 2) *incremental* compositional symbolic execution is a dramatic booster for targeting all k-way interaction trees with the reuse mechanism compared with the non-incremental compositional symbolic execution only.

Next we present a more detailed discussion for each of these three techniques: the coverage criteria, sampling techniques and integration testing techniques.

6.1.1 Coverage Criteria

Coverage criteria can guide testing efforts to special properties of a software systems with a quantified numbers so that defects can be detected in less time. For SPLs, variability is a core property and a source of an exponential number of products. With the variability models of SPLs, variability can be represented as optional features or grouped features with lower and higher bounds. Usually different features may involve constraints like *requires* and *excludes*. Although our work in Chapter 3 can be generalized to all variability models, we focused on OVM models to show the feasibility. We quantified the interactions and constraints among features. More specifically, we translated an OVM model to a relational model, which was then transformed to a CCIT model. With a CCIT model, we

can sample interactions incrementally from 2-, 3-, ... until t-way, and enforce constraints incrementally. With these interactions- and constraints-sensitive coverage criteria, we can drive testing efforts to these areas with a quantitative guideline.

6.1.2 Sampling Techniques

After we developed the coverage criteria, including interaction- and constraints-sensitive criteria, we focused on one difficulty in Chapter 4, i.e. how to handle constraints during a CA construction. Prior to our work, there were few efficient and effective CA generators with clearly described algorithms to target constraints. Traditionally, researchers believed that CA generators will be slower due to added constraints. In this dissertation, we proposed several related algorithms with a detailed explanation and evaluation, and we found that CA generators could be much faster with no loss in quality for covering arrays.

First, we handled constraints by integrating two constraint solvers, zChaff and MiniSat, with two CA generators, a Simulated Annealing generator and an AETG-like generator. We called the AETG-like algorithm AETG-SAT, which is a base line for the following variant algorithms. More specifically, we checked if a partial row is consistent with constraints in several steps. First, we encoded partial row and the user-specified constraints as a CNF formula, and then passed the formula to a solver. Based on the returned true/false results, we decided to extend the partial row for the true result or to replace it with another partial row for the false result correspondingly. As conjectured by researchers, the performance for handling constraints was worse than the one without considering constraints. But AETG-SAT's performance was close to the algorithm without considering constraints, which held promise for the further investigation.

Second, we observed that after we fed a partial row to a SAT solver, it returned a complete model if the partial row was satisfiable. For such a complete model, some

bindings are decided by a BCP procedure based on the partial row. All these bindings must be kept in the remaining partial row constructions by the CA generator. Some are positive bindings called a *Must* set and some are negative bindings called a *May* set. Both sets can speed the construction effort of AETG-SAT. When a factor corresponds to the factor from the *Must* set, then we can bind the factor with the value from the *Must* without any further best-value and satisfiability checking. When a factor is from the *May* set, we can then reduce the choices of that factor for best value checking. We call the variant algorithm, AETG-Hist. From experiments with four real large software systems and 30 synthesized models, we observed that AETG-Hist can produce high-quality samples with the same cost as unconstrainted samples. This result is a dramatic improvement over AETG-SAT.

Third, we observed another basic fact – that both an SAT solver and a CA generator search a model and a row correspondingly in a same combinatorial space for a CCIT problem. This fact suggests that we can save efforts for a row construction of a CA generator by exploiting more aggresively the returned model from an SAT solver. We developed an algorithm, AETG-Threshold, for exploiting a model from an SAT solver to extend a partial row to be a complete row in one step. This saves the construction efforts of AETG-SAT dramatically. Based on our empirical study, we observed that the performance of AETG-Threshold is fastest in terms of construction time. We designed another variant algorithm, AETG-Hist-Threshold, to combine AETG-Hist and AETG-Threshold. The purpose is to utilize AETG-Hist for obtaining a high quality covering array and to utilize AETG-Threshold for accelerating the construction speed of a covering array. From the study, AETG-Hist-Threshold presented the best quality of covering arrays with the second-fastest speed.

6.1.3 Integration Testing SPLs

After designing a series of incrementally improved algorithms for sampling the space of an SPL, we developed a hierarchical reuse mechanism for integration testing an SPL in Chapter 5. In this method, we observed that an interaction may appear in many larger interactions of an SPL. By digging into the inner structure of an interaction, we model it as a directed graph. Each node represents a feature, and each edge represents a directed data dependence relation between two features. For generating test cases to trigger an interaction, we need to know the directions among features in the interaction. In the method, we decomposed a directed graph with multiple trees, where each tree represents an atomic directed interaction pattern for an interaction. We organize all interaction trees as an interaction hierarchy including 1-, 2-, 3-, . . . , and k-way interaction trees. Higher level interaction trees can be composed with lower interaction trees, which illustrates the bottom-up reuse mechanism. For each tree, we proposed a compositional symbolic execution to compute the summaries by composing summaries of lower level interaction trees. In the feasibility study of two subjects, the static method was as much as four times faster compared with the directed symbolic execution method,

We also compared the number of interactions with/without directions. In the experiment, we reduced a number of interactions from a range between 12.2 and 99.9% for the interactions without directions. Although more extensive subjects are needed to verify the reduction, our results definitely showed that a large number of undirected interactions are infeasible, and thus we do not need to test them in the traditional testing methods.

Next we propose three areas for our future work along the line of sampling and integration testing an SPL.

6.2 Future Work

There are two extensions for both directed symbolic execution and our proposed compositional integration testing to reduce the number of summaries. There are also three future directions for applications of both the sampling and integration testing techniques: exploring collected paths, integration testing a sample and the bug isolation. We discuss extensions first.

6.2.1 Extension of Integration Testing Methods

For complete paths of 2-, 3-, ... and k-way interaction trees in Chapter 5, they are supposed to trigger def-use pairs across features, but not all complete paths for these interaction trees consist of *defs* and corresponding *uses*. By removing this type of complete path, we can reduce the summary size without losing the effectiveness of triggering def-use pairs. More specifically, we can perform the optimization below.

Rather than combining all of the summaries into a summary of the explored behavior of a feature. We can distinguish three sets:

1. $Summary_{def}$: the paths which traversed a def involved in a feature interaction,
2. $Summary_{use}$: the paths which traversed a use involved in a feature interaction,
3. $Summary_{ot}$: other paths that are traversed, and
4. unknown = $\neg(Summary_{def} \cup Summary_{use} \cup Summary_{ot})$.

Note here that $Summary_{def} \cup Summary_{use} = \emptyset$ need not be true. We can have paths which involve both defs and uses. The interesting bit here is that when composing summaries in an interaction tree one can compose the def summaries of one feature with the use summaries of another to expose interactions. In particular, there is no need to

consider the use-use, ot-*, use-def, or *-ot summary compositions (where * means any of the summaries). This will also lead to improved performance with no loss in accuracy. Note further that we must allow for the fact that "unknown" may involve def/uses of interest so when generating tests we have to consider def-unknown, unknown-use, and unknown-unknown as potentially interesting regions of behavior.

Similarly, we can perform an optimization for the directed symbolic execution. The directed symbolic execution can explore only those paths that contain def/use statements that are involved in chains of a feature interaction. Since we have the dependence analysis results it is easy to mark the statements in the program and propagate that information up to branches. We then annotate these branches so that symbolic execution can skip branches that cannot lead to def/uses of interest. This will also improve performance with no loss in accuracy.

Finally, a case study is needed to observe the effectiveness of these two optimizations for both techniques correspondingly.

6.2.2 Exploitation of Collected Paths

Chapter 5 defines three possible paths: complete, exception and incomplete paths. Our compositional symbolic execution utilizes only complete paths, which finish the execution without exceptions under the depth limitation over the symbolic execution engine. It is useful to exploit complete paths further and to use the remaining two types of paths for other testing tasks.

For complete paths of 1-, 2-, ... and k-way interaction trees, it is natural to feed them into an SMT solver for generating test cases. For each tree, all generated test cases are distinguished with each other for covering different paths. There are three applications of these test cases. First, these test cases can be run to collect real def-use pairs

accross feature boundaries, which can further validate the effectiveness of our proposed integration testing technique besides the feasibility in the dissertation. Second, they can be used to enhance an existing test suite for single methods as unit-level test cases, multiple methods as integration-level test cases and complete products as system-level test cases. Third, in Chapter 5, complete paths are not classified into exception paths only because they return normally, which is a basic correctness criterion. With more refined domain-related oracles, these complete paths and corresponding test cases may detect more interaction faults.

For exception paths of a feature, as mentioned in 5.4, they may indicate faults in the feature. Because we compute paths of a feature for any possible state on entry to the feature, it is possible some exception paths are infeasible when composed with other features. With the confirmation of infeasibility for some exception paths, testing resources can be allocated to analyze other feasible bugs. Currently the compositional symbolic execution tool only computes complete paths for 1-way interaction tree, but we can easily extend the tool to compute exception paths. For 2- and higher-way interaction trees, we can compute their exception paths by composing complete summaries of all features other than the root feature, for which exception paths should be used. Besides the extension of the hierarchical composition tool, an empirical study over more subjects is useful to quantify the incremental reduced number of exception paths for a feature during the composition procedure.

For incomplete paths of a given feature, we recoganize them as part of the unknown-space of that feature. The whole unknown space of a tree may include other paths which have not been fully explored due to the time limitation for a big depth setup of an symbolic execution engine. The unknown space should be the focus for detecting extra bugs. Because of the accessibility of these collected incomplete paths, we can drive other cheaper testing methods to the partial unknown space first. Random testing and

search-based testing techniques are perfect candidates for constructing effective test cases to probe the partial unknown space. These test cases must satisfy conditions represented in the partial unknown space. In this dissertation, we focus on incomplete paths for single features, and we can also compute incomplete paths for 2- and higher-way interaction trees by composing incomplete and complete summaries of involved features.

6.2.3 Mixture of Sampling and Integration Testing

For integration testing an SPL, one limitation is the scalability problem related to a huge number of interaction trees. Based on the conducted feasibility experiment, after reaching a small k-way level, it will take much longer time to compose all (k+1)-way interaction trees. To tackle the scalability problem, we can mix the sampling and integration testing together. Both techniques have orthogonal properties, and by integration testing a sample, we compose summaries only for involved interaction trees to fulfill specific coverage criteria.

With a sampling technique, we can get a subset of all products with a predefined interaction coverage. With the integration testing technique, we can do integration testing of all interaction trees in the sample incrementally. More specifically, we construct a smaller size of FDG for the sample. Driven by the customized interaction tree hierarchy, we do the integration testing over interaction trees bottom-up with the reuse mechanism. A sample is much smaller than the whole space of an SPL, so this mixed technique may have wider applications than the original method.

6.2.4 Bug Isolation

During the development of SPLs, methods related to bug isolation are useful for helping tester engineers to locate bugs quickly for a runtime error. Usually these bugs reflect

complicated interactions among multiple features, and unit testing methods are not helpful for detecting these type of bugs. Chapter 5 discussed an integration testing method over SPLs from the feature interaction perspective. Our integration testing method can target this type of feature-interaction bugs.

More specifically, given a runtime error occuring in a feature, f, we can customize the FDG to be FDG' by keeping only features reaching to f. Note we do not consider a loop scenario starting and ending at f because we compute summaries of features with the consideration of any possible state on entry to these features. After constructing a potential smaller FDG, we then collect all 2-, 3-, . . . , and k-way interaction trees sinking at f. Incrementally, we can detect if one or more of 2-, 3-, . . . until k-way trees can expose the same observed bad behavior.

There are two benefits for our method. First, because we use an incremental strategy, we can expose the bad behavior with the smallest scope. Second, because we can collect more than one tree (if applicable), then we can locate bugs with more confidence.

Bibliography

[1] Apache Software Foundation. Apache HTTP sever. http://httpd.apache.org/docs/2.2/mod/quickreference.html, 2007. 4.6.1

[2] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 261–272, New York, NY, USA, 2008. ACM. 2.4.3

[3] E. S. B. Hnich, S. Prestwich. Constraint-based approaches to the covering test problem. In *CSCLP04 Special Volume of Lecture Notes in Artificial Intelligence*, 2005. 2.3

[4] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society. 2.1.1

[5] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 30(6):2004, 2004. 2.1.1

[6] D. S. Batory. Feature models, grammars, and propositional formulas. In *SPLC*, pages 7–20, 2005. 2.1.1

[7] J. Belt, Robby, and X. Deng. Sireum/Topi LDP: a lightweight semi-decision procedure for optimizing symbolic execution-based analyses. In *Proceedings of the the 7th*

joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ESEC/FSE '09, pages 355–364, New York, NY, USA, 2009. ACM. 2.4.2

[8] D. Benavides, S. Segura, P. Trinidad, and A. R. Cortés. FAMA: tooling a framework for the automated analysis of feature models. In *In Proceeding of the First International Workshop on Variability Modelling of Softwareintensive Systems (VAMOS*, pages 129–134, 2007. 1.2.4, 2.1.1

[9] A. Bertolino and S. Gnesi. PLUTO: a test methodology for product families. In *Software Product-Family Engineering (PFE)*, pages 181–197, 2003. 2.5.2

[10] R. Brownlie, J. Prowse, and M. S. Phadke. Robust testing of AT&T PMX/StarMAIL using OATS. *AT& T Technical Journal*, 71(3):41–47, 1992. 2.2.1

[11] R. C. Bryce, C. J. Colbourn, and M. B. Cohen. A framework of greedy methods for constructing interaction test suites. In *Proceedings of the International Conference on Software Engineering*, pages 146–155, May 2005. 4.1

[12] I. Cabral, M. B. Cohen, and G. Rothermel. Improving the testing and testability of software product lines. In *Proceedings of the 14th international conference on Software product lines: going beyond*, SPLC'10, pages 241–255, Berlin, Heidelberg, 2010. Springer-Verlag. 2.5.1

[13] C. Cadar, D. Engler, and P. Boonstoppel. Attacking path explosion in constraint-based test generation. In *ETAPS Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008) Budapest, Hungary, March-April 2008*, 2008. 2.4.1

[14] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Proceeding of the 33rd international conference on Software engineering*, ICSE '11, pages 1066–1071, New York, NY, USA, 2011. ACM. 2.4

[15] Choco. http://www.emn.fr/z-info/choco-solver/, 2011. 1.2.4

[16] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. volume 19, pages 7–34, Hingham, MA, USA, July 2001. Kluwer Academic Publishers. 2.3

[17] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic model checking of software product lines. In *Proceeding of the 33rd international conference on Software engineering*, ICSE '11, pages 321–330, New York, NY, USA, 2011. ACM. 2.5.2

[18] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 335–344, New York, NY, USA, 2010. ACM. 2.5.2

[19] P. Clements and L. M. Northrop. *Software product lines: practices and patterns*. Addison Wesley, 2001. 1, 1.1

[20] CMU. http://www.sei.cmu.edu/productlines/. 1

[21] A. Cockburn. *Writing effective use cases*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 2000. 2.5.2

[22] 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. 2.2.1, 2.2.5, 4.1

[23] M. B. Cohen, C. J. Colbourn, P. B. Gibbons, and W. B. Mugridge. Constructing test suites for interaction testing. In *Proceedings of the International Conference on Software Engineering*, pages 38–48, May 2003. 2.2.3, 4.6.5

[24] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling. Augmenting simulated annealing to build interaction test suites. In *14th IEEE International Symposium on Software Reliability Engineering*, pages 394–405, November 2003. 2.2.3

[25] M. B. Cohen, M. B. Dwyer, and J. Shi. Coverage and adequacy in software product line testing. In *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, ROSATEA '06, pages 53–63, New York, NY, USA, 2006. ACM. 2.1.1, 3, 1

[26] M. B. Cohen, M. B. Dwyer, and J. Shi. Exploiting constraint solving history to construct interaction test suites. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 121–132, Washington, DC, USA, 2007. IEEE Computer Society. 1.2.2, 2.2.3, 4, 1

[27] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *ISSTA*, pages 129–139, 2007. 1.2.2, 2.2, 2.2.3, 4, 1, 4.6.5

[28] M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: a greedy approach. *IEEE Trans. Softw. Eng.*, 34(5):633–650, 2008. 1.2.2, 2.2.3, 4, 1

[29] C. J. Colbourn, M. B. Cohen, and R. C. Turban. A deterministic density algorithm for pairwise interaction coverage. In *IASTED Proceedings of the International Conference on Software Engineering*, pages 345–352, February 2004. 2.2.5, 4.1

[30] Communication Research Center Canada. http://www.crc.gc.ca/en/html/crc/home/research/satcom/rars/sdr/products/scari_open/scari_open. 5.5.1, 5.5.1

[31] Congit Software. Congitproduct conguration engine. http://www.configit-software.com/, 2005. 2.1.1

[32] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *Proc. 30th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 281–290. ACM, May 2008. 2.4.3

[33] CVC3. http://www.cs.nyu.edu/acsys/cvc3/, 2011. 1.2.4

[34] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In *Software Product Lines: Third International Conference, SPLC 2004*, pages 266–283. Springer-Verlag, 2004. 2.1.1, 2.1.1

[35] K. Czarnecki, S. Helson, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process : Improvement and Practice*, 10(1):7–29, 2005. 2.1.1

[36] K. Czarnecki and C. H. P. Kim. Cardinality-based feature modeling and constraints: a progress report. In *International Workshop on Software Factories at OOPSLA'05*, San Diego, California, USA, 2005. ACM, ACM. 2.1.1, 2.1.1

[37] J. Czerwonka. Pairwise testing in real world. In *Pacific Northwest Software Quality Conference*, pages 419–430, October 2006. 4.1

[38] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. volume 5, pages 394–397, New York, NY, USA, July 1962. ACM. 2.3, 2.3.1, 2.3.3

[39] M. Davis and H. Putnam. A computing procedure for quantification theory. In *Journal of the ACM* , pages 7(3):201–215, July 1960. 2.3

[40] N. Eén and N. Sörensson. An extensible sat-solver. In *SAT*, pages 502–518, 2003. 1.2.4, 2.3

[41] N. Eén and N. Sörensson. Translating pseudo-boolean constraints into sat. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 2(1-4):1–26, 2006. 2.3

[42] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM. 2.4.3

[43] E. Falkenauer. *Genetic algorithms and grouping problems*. John Wiley & Sons, Inc., New York, NY, USA, 1998. 2.2.3

[44] D. Fischbein, S. Uchitel, and V. Braberman. A foundation for behavioural conformance in software product line architectures. In *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, ROSATEA '06, pages 39–48, New York, NY, USA, 2006. ACM. 2.5.2

[45] K. D. Forbus and J. de Kleer. *Building Problem Solvers*. MIT Press, 1993. 2.1.1

[46] Free Software Foundation. GNU 4.1.1 manpages. http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/, 2005. 4.6.1, 4.6.1.2

[47] D. Ganesan, M. Lindvall, D. McComas, M. Bartholomew, S. Slegel, and B. Medina. Architecture-based unit testing of the flight software product line. In J. Bosch and J. Lee, editors, *Software Product Lines: Going Beyond*, volume 6287 of *Lecture Notes in Computer Science*, pages 256–270. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-15579-6_18. 2.1

[48] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering*, 16(1):61–102, 2011. 2.2.3

[49] B. Geppert, J. Li, F. Rößler, and D. M. Weiss. Towards generating acceptance tests for product lines. In *Software Reuse: Methods, Techniques and Tools (ICSR)*, pages 35–48, 2004. 2.5.2

[50] F. Glover and M. Laguna. *Tabu search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997. 2.2.3

[51] GNU Radio. http://gnuradio.org/redmine/wiki/gnuradio. 5.5.1, 5.5.1

[52] P. Godefroid. Compositional dynamic test generation. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–54, New York, NY, USA, 2007. ACM. 2.4.1

[53] P. Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 47–54, 2007. 2.4.4

[54] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: a survey. *Software Testing, Verification and Reliability*, 15(3):167–199, 2005. 2.2.5

[55] M. L. Griss, J. Favaro, and M. d. Alessandro. Integrating Feature Modeling with the RSEB. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, pages 76–, Washington, DC, USA, 1998. IEEE Computer Society. 2.1.1

[56] H. Gustavsson and U. Eklund. Architecting automotive product lines: Industrial practice. In J. Bosch and J. Lee, editors, *Software Product Lines: Going Beyond*, volume

6287 of *Lecture Notes in Computer Science*, pages 92–105. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-15579-6_7. 2.1

[57] H. R. Andersen. An introduction of binary decision diagrams. Department of Information Technology, Technical University of Denmark, Lyngby, Denmark. Lecture notes for 49285 Advanced Algorithms E97, 1997. 2.1.1

[58] R. L. Halpert. Static lock allocation. Master's thesis, McGill University, April 2008. 5.5.2

[59] A. Hartman. Software and hardware testing using combinatorial covering suites. In *Graph Theory, Combinatorics and Algorithms: Interdisciplinary Applications*, pages 327–266, 2005. 2.2.2

[60] A. Hartman and L. Raskin. Problems and algorithms for covering arrays. *Discrete Math*, 284:149 – 156, 2004. 2.2.2

[61] S. A. Hendrickson and A. van der Hoek. Modeling product line architectures through change sets and relationships. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 189–198, Washington, DC, USA, 2007. IEEE Computer Society. 2.1

[62] W. A. Hetrick, C. W. Krueger, and J. G. Moore. Incremental return on incremental investment: Engenio's transition to software product line practice. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 798–804, New York, NY, USA, 2006. ACM. 1.1

[63] B. Hnich, S. Prestwich, E. Selensky, and B. Smith. Constraint models for the covering test problem. *Constraints*, 11:199–219, 2006. 2.2.4

[64] B. Hnich, S. D. Prestwich, and E. Selensky. Constraint-based approaches to the covering test problem., 2004.

[65] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997. 4.6.1, 4.6.1.1

[66] G. J. Holzmann. On-the-fly, LTL model checking with SPIN: Man pages. http://spinroot.com/spin/Man/index.html, 2006. 4.6.1.1

[67] S. Huang, M. B. Cohen, and A. M. Memon. Repairing GUI test suites using a genetic algorithm. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 245–254, Washington, DC, USA, 2010. IEEE Computer Society. 2.2.1

[68] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11:256–290, April 2002. 2.5.2, 4.2.1

[69] JaCoP. http://jacop.osolpro.com/, 2011. 2.1.1

[70] A. Jansen, R. Smedinga, J. van Gurp, and J. Bosch. Feature-based product derivation: Composing features. In *IEE Proceedings Software - special issue on Software Engineering*, August 2004. 2.1

[71] M. Jaring and J. Bosch. Expressing product diversification – categorizing and classifying variability in software product family engineering. *International Journal of Software Engineering and Knowledge Engineering*, 14(5):449–470, 2004. 5.2

[72] S. Jarzabek, W. C. Ong, and H. Zhang. Handling variant requirements in domain modeling. *J. Syst. Softw.*, 68:171–182, December 2003. 2.1.1

[73] JavaBDD. http://javabdd.sourceforge.net/, 2011. 2.1.1

[74] R. J. B. Jr. and R. C. Schrag. Using CSP look-back techniques to solve real world sat instances. 1997. 2.3, 2.3.3

[75] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. 1990. 2.1.1

[76] C. Kastner, T. Thum, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. FeatureIDE: A tool framework for feature-oriented software development. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 611–614, Washington, DC, USA, 2009. IEEE Computer Society. 2.1.1

[77] R. Kauppinen, J. Taina, and A. Tevanlinna. Hook and template coverage criteria for testing framework-based software product families. In *Proceedings of the International Workshop on Software Product Line Testing*, pages 7–12, August 2004. 2.5.1

[78] C. H. P. Kim, D. S. Batory, and S. Khurshid. Reducing combinatorics in testing product lines. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 57–68, New York, NY, USA, 2011. ACM. 2.5.2, 5.5.1, 5.5.1

[79] C. H. P. Kim, E. Bodden, D. S. Batory, and S. Khurshid. Reducing configurations to monitor in a software product line. In *Proceedings of the First international conference on Runtime verification*, RV'10, pages 285–299, Berlin, Heidelberg, 2010. Springer-Verlag. 2.5.2

[80] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976. 2.4

[81] J. Kiniry. Reasoning about feature models in higher-order logic. In *Proceedings of the 11th International Software Product Line Conference, SPLC 07. IEEE Computer Society*, 2007. 2.1.1

[82] D. Kroening and O. Strichman. *Decision procedures: an algorithmic point of view*. Springer Publishing Company, Incorporated, 1st edition, 2008. 2.4.2, 4.2.1

[83] D. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, 2004. 2.2.1

[84] T. Larrabee. Test pattern generation using boolean satisfiability. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 11(1):4 –15, jan 1992. 2.3

[85] K. Larsen and B. Thomsen. A modal process logic. In *Logic in Computer Science, 1988. LICS '88., Proceedings of the Third Annual Symposium on*, pages 203 –210, jul 1988. 2.5.2

[86] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. Ipog: A general strategy for t-way software testing. In *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, pages 549–556, Washington, DC, USA, 2007. IEEE Computer Society. 2.2.5

[87] R. E. Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In *Proc. Conf. on Generative and Component-Based Soft. Eng*, pages 10–24. Springer, 2001. 5.5.1, 5.5.1

[88] R. Mandl. Orthogonal latin squares: An application of experiment design to compiler testing. *Communications of the ACM*, 28(10):1054–1058, October 1985. 2.2.1

[89] M. Mannion. Using first-order logic for product line model validation. In *Proceedings of the Second International Conference on Software Product Lines*, SPLC 2, pages 176–187, London, UK, UK, 2002. Springer-Verlag. 2.1.1

[90] J. P. Marques-Silva and K. A. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999. 2.3

[91] J. D. McGregor. Testing a software product line. Technical report, Carnegie Mellon, Software Engineering Institute, December 2001. 2.2.1, 2.5.1

[92] Microsoft. Pex. http://research.microsoft.com/en-us/projects/pex/, 2011. 2.4.3

[93] Microsoft. Yogi. http://research.microsoft.com/en-us/projects/yogi/, 2011. 2.4.3

[94] MiniSat. http://minisat.se/MiniSat.html, 2011. 4.6.5

[95] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM. 1.2.4, 2.3

[96] Mozilla Organization. Bugzilla. http://www.bugzilla.org/docs/tip/html/, 2007. 4.6.1, 4.6.1.4

[97] H. Muccini and A. van der Hoek. Towards testing product line architectures. *Electr. Notes Theor. Comput. Sci.*, 82(6):99–109, 2003. 2.5.1

[98] K. J. N. and P. R. J. O. *Constructing covering designs by simulated annealing*. Technical Report B10, Helsinki University of Technology, Digital Systems Laboratory, Otaniemi, Finland, 1993. 2.2.3

[99] G.-J. Nam, K. Sakallah, and R. Rutenbar. A new FPGA detailed routing approach via search-based Boolean satisfiability. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 21(6):674 –684, jun 2002. 2.3

[100] NASA. SPF. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc, 2011. 2.4.3, 5.5.2

[101] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43:11:1–11:29, February 2011. 2.2.5

[102] K. Nurmela. Upper bounds for covering arrays by tabu search. *Discrete Applied Mathematics*, 138(1-2):143–152, 2004. 2.2.3

[103] S. Oster, F. Markert, and P. Ritter. Automated incremental pairwise testing of software product lines. In *Proceedings of the 14th international conference on Software product lines: going beyond*, SPLC'10, pages 196–210, Berlin, Heidelberg, 2010. Springer-Verlag. 2.2.4

[104] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, 1988.

[105] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184, 1984. 5.3

[106] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 226–237, New York, NY, USA, 2008. ACM. 2.4.1, 2.4.3, 2.4.4, 2.4.1

[107] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 504–515, New York, NY, USA, 2011. ACM. 2.4.1

[108] J. F. Peters and W. Pedrycz. *Software engineering: an engineering approach*. 1999. 1.1

[109] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software product line engineering: foundations, principles and techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. (document), 1, 1.2.1, 2.1, 2.1.1, 2.1.1, 3.1

[110] K. Pohl and A. Metzger. Software product line testing. *Commun. ACM*, 49:78–81, December 2006. 2.1.1

[111] C. S. Păsăreanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf.*, 11:339–353, October 2009. 2.4.3

[112] Pure::Viriants. http://www.pure-systems.com/, 2011. 2.1.1

[113] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 75–86, New York, NY, USA, 2008. ACM. 2.2.1

[114] S. Reis, A. Metzger, and K. Pohl. Integration testing in software product line engineering: a model-based technique. In *Proceedings of the 10th international conference on Fundamental approaches to software engineering*, FASE'07, pages 321–335, Berlin, Heidelberg, 2007. Springer-Verlag. 2.1.1, 2.5.2

[115] T. Reps and G. Rosay. Precise interprocedural chopping. In *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 41–52, 1995. 5.1, 5.2

[116] M. Riebisch, K. Bllert, D. Streitferdt, and I. Philippow. Extending feature diagrams with uml multiplicities, 2002. 2.1.1

[117] Sat4J. http://www.sat4j.org/, 2011. 2.1.1, 2.3

[118] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: a survey and a formal semantics. In *Proceedings of the 14th IEEE International Requirements Engineering Conference*, pages 136–145, Washington, DC, USA, 2006. IEEE Computer Society. 2.1.1

[119] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *In Proceedings of the IEEE Symposium on Security and Privacy*, 2010. 2.4.3

[120] D. Shannon, S. Hajra, A. Lee, D. Zhan, and S. Khurshid. Abstracting symbolic execution with string analysis. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 13–22, Washington, DC, USA, 2007. IEEE Computer Society. 2.4.2

[121] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In *Proceeding of the 33rd international conference on Software engineering*, ICSE '11, pages 461–470, New York, NY, USA, 2011. ACM. 2.1.1

[122] J. Shi, M. B. Cohen, and M. B. Dwyer. Integration testing of software product lines using compositional symbolic execution. In *Proceedings of the 10th international conference on Fundamental approaches to software engineering*, 2012. 1

[123] W. J. Slegers. Building automotive product lines around managed interfaces. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 257–264, Pittsburgh, PA, USA, 2009. Carnegie Mellon University. 2.1

[124] Sox . http://sox.sourceforge.net/. 5.5.1, 5.5.1

[125] SPLC History. http://splc.net/history.html, 2011. 2.1

[126] Stanford. Klee. http://klee.llvm.org/, 2011. 2.4.3

[127] P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. Number UCB/ERL M92/112, 1992. 2.3

[128] B. Stevens and E. Mendelsohn. New recursive methods for transversal covers. *Journal of Combinatorial Designs*, (7):185203, 1999. 2.2.2

[129] P. Stoll, L. Bass, E. Golden, and B. E. John. Supporting usability in product line architectures. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 241–248, Pittsburgh, PA, USA, 2009. Carnegie Mellon University. 2.1

[130] V. Stricker, A. Metzger, and K. Pohl. Avoiding redundant testing in application engineering. In *Proceedings of the 14th international conference on Software product lines: going beyond*, SPLC'10, pages 226–240, Berlin, Heidelberg, 2010. Springer-Verlag. 2.1.1

[131] J. Sun, H. Zhang, and H. Wang. Formal semantics and verification for feature modeling. Technical report, Washington, DC, USA, 2005. 2.1.1

[132] K. C. Tai and Y. Lei. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering*, 28(1):109–111, 2002. 2.2.5

[133] A. Tevanlinna. Product family testing with RITA. In *Proceedings of the Eleventh Nordic Workshop on Programming and Software Development Tools and Techniques (NW-PER'2004)*, pages 251–265, August 2004. 2.5.1

[134] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 95–104, New York, NY, USA, 2007. ACM. 2.1.1

[135] T. Thüm, D. S. Batory, and C. Kästner. Reasoning about edits to feature models. In *ICSE*, pages 254–264, 2009. 2.1.1

[136] Y.-W. Tung and W. Aldiwan. Automating test case generation for the new generation mission software system. In *Aerospace Conference Proceedings, 2000 IEEE*, volume 1, pages 431 –437 vol.1, 2000. 2.2.5, 4.1

[137] UC Berkeley. Bitblaze. http://bitblaze.cs.berkeley.edu/, 2011. 2.4.3

[138] UC Berkeley. Crest. http://code.google.com/p/crest/, 2011. 2.4.3

[139] UIUC. Cute and jcute. http://osl.cs.uiuc.edu/ ksen/cute/, 2011. 2.4.3

[140] E. Uzuncaova, D. Garcia, S. Khurshid, and D. Batory. Testing software product lines using incremental test generation. In *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, pages 249–258, Washington, DC, USA, 2008. IEEE Computer Society. 2.5.2

[141] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Intl. Conf. on Compiler Construction (2000), Springer-Verlag (LNCS)*, 2000. 5.5.2

[142] J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *In Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA01*, pages 45–54. IEEE Computer Society, 2001. 2.1.1

[143] Variability Model Editor. http://www.sse.uni-due.de/wms/en/index.php?go=256, 2011. 2.1.1

[144] M. Vierhauser, P. Grünbacher, A. Egyed, R. Rabiser, and W. Heider. Flexible and scalable consistency checking on product line variability models. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 63–72, New York, NY, USA, 2010. ACM. 2.1.1

[145] H. S. Wang, S. R. Hsu, and J. C. Lin. A generalized optimal path-selection model for structural program testing. *Journal of Systems and Software*, pages 55–63, 1989. 2.5.2

[146] A. W. Williams. Determination of test configurations for pair-wise interaction coverage. In *Proceedings of the IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems: Tools and Techniques*, TestCom '00, pages 59–74, Deventer, The Netherlands, The Netherlands, 2000. Kluwer, B.V. 2.2.5

[147] J. Yan and J. Zhang. An efficient method to generate feasible paths for basis path testing. *Inf. Process. Lett.*, 107:87–92, July 2008. 2.5.1

[148] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *In Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 243–257, 2006. 2.4.3

[149] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. In *International Symposium on Software Testing and Analysis*, pages 45–54, July 2004.

[150] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 31(1):20–34, 2006. 2.2.1

[151] X. Yuan, M. B. Cohen, and A. M. Memon. GUI interaction testing: Incorporating event context. *IEEE Transactions on Software Engineering*, 37:559–574, 2011. 2.2.1

[152] Z3. http://research.microsoft.com/en-us/um/redmond/projects/z3/, 2011. 1.2.4

[153] zChaff. http://www.princeton.edu/ chaff/zchaff.html, 2004. 2.3.1

[154] A. Zeller. *Why programs fail, second edition: a guide to systematic debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2009. 1.1

[155] H. Zhang. SATO: an efficient propositional prover. In *Proceedings of the 14th International Conference on Automated Deduction*, CADE-14, pages 272–275, London, UK, 1997. Springer-Verlag. 2.3

[156] L. Zhang. Search for truth: techniques for satisfiability of boolean formulas. In *PhD Thesis 2003*, 2003. 2.3.2