Pós-Graduação em Ciência da Computação

Sabrina de Figueirêdo Souto

ADDRESSING HIGH DIMENSIONALITY AND LACK OF FEATURE MODELS IN TESTING OF SOFTWARE PRODUCT LINES

Ph.D. Thesis

Federal University of Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE

2015

Sabrina de Figueirêdo Souto

ADDRESSING HIGH DIMENSIONALITY AND LACK OF FEATURE MODELS IN TESTING OF SOFTWARE PRODUCT LINES

A Ph.D. Thesis presented to the Center for Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Philosophy Doctor in Computer Science.

Advisor: *Marcelo d'Amorim*

RECIFE
2015

Catalogação na fonte
Bibliotecária Joana D'Arc Leão Salvador CRB4-532

S728a Souto, Sabrina de Figueirêdo.
Addressing high dimensionality and lack of feature models in testing of software product lines / Sabrina de Figuerêdo Souto. – Recife: O Autor, 2015.
104 f.: fig., tab.

Orientador: Marcelo Bezerra d'Amorim.
Tese (Doutorado) – Universidade Federal de Pernambuco. CIN, Ciência da Computação, 2015.
Inclui referências.

1. Engenharia de software. 2. Software - testes. 3. . I. Amorim, Marcelo Bezerra d' (Orientador). II. Titulo.

005.1 CDD (22. ed.) UFPE-MEI 2015-094

Tese de doutorado apresentada por **Sabrina de Figueirêdo Souto** ao programa de Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título **Addressing High Dimensionality and Lack of Feature Models in Testing of Software Product Lines**, orientada pelo **Prof. Marcelo d'Amorim** e aprovada pela banca examinadora formada pelos professores:

Prof. Paulo Henrique Monteiro Borba
Centro de Informática/UFPE

Prof. Juliano Manabu Yoda
Centro de Informática/UFPE

Prof. Fernando José Castor de Lima Filho
Centro de Informática/UFPE

Prof. Marco Tulio Valente
Departamento de Computação/UFMG

Prof. Rohit Gheyi
Departamento de Sistemas e Computação/UFCG

RECIFE
2015

To my family.

Acknowledgements

Depois de mais de 200 mil quilômetros viajando entre Campina Grande e Recife durante os 4 anos do doutorado, consegui concretizar mais um sonho. Porém, sem a ajuda de algumas pessoas não teria conseguido chegar até aqui, seguem meus sinceros agradecimentos:

- Primeiramente a Deus, por me sustentar e me dar muito mais do que mereço, por me capacitar a perseverar sempre, a Ele toda honra e toda glória;
- Ao meu marido (Roberto) e meu filho (Tiago), por me amarem o suficiente para tolerar minhas ausências, por me encorajar e apoiar. Em especial, agredeço a Roberto pela parceria e colaboração, sem as quais eu não teria conseguido;
- Aos meus pais, a vóvó Lála e a Preta, e aos meus sogros, por me encorajar e sempre me proporcionar todas as condições necessárias para que eu pudesse terminar o doutorado;
- Ao meu orientador, professor Marcelo d'Amorim, por me guiar e apoiar nesta difícil caminhada, pela preocupação com a minha formação, e por todos os ensinamentos que levarei pelo restante da vida. Agradeço ainda por ter me proporcionado preciosas colaborações com alunos e professores de autoridade no assunto dessa pesquisa, em especial com o professor Darko Marinov (Universidade de Illinois - EUA), por ter fornecido motivações práticas para direcionar esse trabalho;
- Ao professor Vinicius Garcia, pelo apoio, acompanhamento e amizade, que me auxiliaram em alguns momentos difíceis dessa caminhada;
- Aos membros da banca da proposta de tese que apresentei no dia 15 de Maio de 2014, composta pelos professores Alessandro Garcia, Paulo Borba, Juliano Iyoda, e Fernando Castor, agradeço pelo excelente feedback que recebi;
- Aos amigos que fiz durante o Doutorado: Francisco Airton, Paulo Barros, Mateus Araújo, Nancy Lira, Patricia Muniz, Liliane Fonseca, Leopoldo Teixeira, Márcio Ribeiro, Paola Accioly, Neto Pires, Thaís Burity, Priscilla Machado, Francisco Soares, Weslley Torres, Andreza Leite, Helaine Solange, Emanoel Barreiros, Fernando Kenji, Elton Alves, João Paulo e Tiago Vieira. Sentirei saudade de todos vocês;
- A todos colegas e amigos do grupo de caronas, que fizeram parte da minha jornada, viajando comigo durante o doutorado, vocês me ajudaram a aguentar o tédio e o cançasso das viagens, e as tornaram mais interessantes e divertidas. Também sentirei falta de vocês;

- Aos professores e funcionários do CIn-UFPE que, de alguma forma, ajudaram na minha formação;
- A FACEPE, pelo apoio financeiro.

A mind that is stretched by a new experience can never go back to its old dimensions.

—OLIVER WENDELL HOLMES, JR.

Resumo

Linhas de Produtos de Software (SPLs) permitem aos engenheiros sistematicamente construirem famílias de produtos de software, definidos por uma combinação única de *features* — incrementos de funcionalidade, melhorando tanto a eficiência do processo de desenvolvimento de software quanto a qualidade do software desenvolvido. Porém, testar esse tipo de sistema é uma tarefa desafiadora, pois requer a execução de cada teste em um número combinatorial de produtos. Chamamos esse problema de *Problema da Alta Dimensionalidade*. Outro obstáculo para o teste de linhas de produtos é a ausência de Modelos de Feature (FMs), dificultando a descoberta das reais causas das falhas nos testes. Chamamos esse problema de *Problema da Falta de Modelo de Features*.

O Problema da Alta Dimensionalidade está associado ao amplo espaço de possibilidades de configurações que uma linha de produtos pode alcançar. Por exemplo, se uma linha de produtos tem *n features* booleanas, então existem 2^n possibilidades de combinações de *features*. Desta forma, para testar esse tipo de sistema, de forma sistemática, pode ser preciso executar cada teste em todas as possíveis combinações, no pior caso. Já o Problema da Falta de Modelo de *Features* está relacionado à falta ou incompletude dos modelos de *feature*. Por esta razão, a falta de FM representa um grande desafio para a descoberta das reais causas para falhas nos testes.

Com o objetivo de resolver esses problemas, propomos duas técnicas leves: `SPLat` e `SPLif`. `SPLat` é uma nova abordagem que ignora configurações irrelevantes: as configurações a serem executadas pelo teste são determinadas durante a execução do teste, através do monitoramento do acesso às opções de configuração. Como resultado, `SPLat` reduz o número de configurações a serem executadas. Consequentemente, `SPLat` é leve comparada a trabalhos anteriores que usam técnicas de análise estática e execuções dinâmicas pesadas. `SPLif` é uma técnica para testar linhas de produtos que não requer modelos de *feature* a priori. A ideia é usar a taxa de execuções de teste que falham e que passam para rapidamente identificar falhas que são indicativas de problemas (no teste ou código), ao contrário de uma falha referente à execução de um teste em uma combinação inconsistente de *features*.

Resultados experimentais mostram que `SPLat` identifica configurações relevantes de forma eficaz, e com uma sobrecarga baixa. Também aplicamos `SPLat` a dois sistemas configuráveis de grande porte (Groupon e GCC), e ele escalou sem muito esforço de engenharia. Resultados experimentais demonstram que `SPLif` é útil e eficaz para rapidamente encontrar testes que falham para configurações consistentes, independente de quão incompleto é o modelo de *features*. Além disso, ainda avaliamos `SPLif` com um sistema configurável de grande porte e extensivamente testado, o GCC, onde `SPLif` ajudou a revelar 5 novas falhas, das quais 3 foram corrigidas após nossos relatórios de falha (*bug reports*).

Palavras-chave: Linhas de Produtos de Software. Sistemas Configuráveis. Teste de Software e Depuração. Feature Model.

Abstract

Software Product Lines (SPLs) allow engineers to systematically build families of software products, defined by a unique combination of features — increments in functionality, improving both the efficiency of the software development process and the quality of the software developed. However, testing these kinds of systems is challenging, as it may require running each test against a combinatorial number of products. We call this problem the *High Dimensionality Problem*. Another obstacle to product line testing is the absence of Feature Models (FMs), making it difficult to discover the real causes for test failures. We call this problem the *Lack of Feature Model Problem*.

The High Dimensionality Problem is associated to the large space of possible configurations that an SPL can reach. If an SPL has n boolean features, for example, there are 2^n possible feature combinations. Therefore, systematically testing this kind of system may require running each test against all those combinations, in the worst case. The Lack of Feature Model Problem is related to the absence of feature models. The FM enables accurate categorization of failing tests as failures of programs or the tests themselves, not as failures due to inconsistent combinations of features. For this reason, the lack of FM presents a huge challenge to discover the true causes for test failures.

Aiming to solve these problems, we propose two lightweight techniques: `SPLat` and `SPLif`. `SPLat` is a new approach to dynamically prune irrelevant configurations: the configurations to run for a test can be determined during test execution by monitoring accesses to configuration variables. As a result, `SPLat` reduces the number of configurations. Consequently, `SPLat` is lightweight compared to prior works that used static analysis and heavyweight dynamic execution. `SPLif` is a technique for testing SPLs that does not require a priori availability of feature models. Our insight is to use a profile of passing and failing test runs to quickly identify test failures that are indicative of a problem (in test or code) as opposed to a manifestation of execution against an inconsistent combination of features.

Experimental results show that `SPLat` effectively identifies relevant configurations with a low overhead. We also apply `SPLat` on two large configurable systems (Groupon and GCC), and it scaled without much engineering effort. Experimental results demonstrate that `SPLif` is useful and effective to quickly find tests that fail on consistent configurations, regardless of how complete the FMs are. Furthermore, we evaluated `SPLif` on one large extensively tested configurable system, GCC, where it helped to reveal 5 new bugs, 3 of which have been fixed after our bug reports.

Keywords: Software Product Lines. Configurable Systems. Software Testing and Debugging. Feature Model.

List of Figures

1.1 Example of code using dynamically bound features. 19
1.2 Example of test that was designed for an specific configuration: $-O3$ and $-fcompare-debug$ 20
1.3 Overview of the solution. 23

2.1 A possible feature diagram of the Graph Library SPL. Source (APEL et al., 2013). 27
2.2 Two compositional implementations of a stack example with three features. Source (KASTNER., 2010). 29
2.3 Annotative implementation using conditional compilation for Graph Library SPL. Source (APEL et al., 2013). 30
2.4 Variability Encoding implementation for Graph Library SPL, conditionally executed code is highlighted. Source (APEL et al., 2013). 31
2.5 Variations intra-method. 32
2.6 Variations inter-method. 33
2.7 Sample SPL Code and Test. 34
2.8 An example test ext-4.c from GCC test suite. 34

3.1 Feature Model. 37
3.2 Notepad SPL and Example Test. 37
3.3 Feature Model Interface 38
3.4 SPLat Algorithm 39
3.5 The distribution of the number of reachable configurations per tests, Figure 3.6 details the distribution of the number of features per tests. 53
3.6 The distribution of the number of features accessed per tests. 53

4.1 Test of `Notepad` that checks the cut and paste functionalities. This test fills the text area with a string, selects the text area, and presses the “cut” button. It then asserts that the text area is indeed empty, presses the “paste” button twice, and checks for the presence of the repeated string. 56
4.2 The `SPLif` Algorithm. 61
4.3 Target SPLs used in `SPLif` evaluation. 63
4.4 Tests and Reachable Configurations. Values in parentheses show the subset of consistent configurations. 63
4.5 Distribution of number of consistent configurations (*CCs*) per number of failing tests (*FTs*). 64

4.6 Counts of passing and failing executions per test and SPL. t_i is the test id. F_i is the number of failures of t_i. P_i is the number of passing executions of t_i. FC_i is the number of consistent configurations in which t_i fails. We omit test entries without failing runs. 64
4.7 Ranking of tests. Column *R* shows the rank of test t_i from Figure 4.6; *S* shows the suspiciousness score of t_i. A row in gray color indicates a test for which at least one failing configuration it reaches is consistent. 65
4.8 Progress inspecting tests and their failing configurations. 67
4.9 The total number of inspections for various techniques, and their reduction compared to two baselines: the number of inspections considering all failures (column *Reduction All(%)*, and the number of inspections considering the *Random* technique (column *Reduction Random(%)* 68

5.1 Example of GCC test using DejaGnu. 75
5.2 Test pr47684.c from gcc.dg test suite. 76
5.3 Artifacts used and produced during `SPLat` execution. 77
5.4 Statistics on Tests. In the left table, column *R* shows the rank of test t_i from Figure 5.4a; *S* shows the suspiciousness score of t_i. A row in gray color indicates a test that requires inspection; a test for which at least one failing configuration it reaches is consistent. In the right table, t_i is the test id; F_i is the number of crashing failures of t_i. P_i is the number of passing executions of t_i; FC_i is the number of consistent configurations in which t_i crashes. We omit test entries without crashing runs. 79
5.5 Configuration inspection progress for GCC. 80
5.6 The total number of inspections for various techniques, and their reduction compared to two baselines: the number of inspections considering all failures (column *Reduction All(%)*, and the number of inspections considering the *Random* technique (column *Reduction Random(%)* 81
5.7 GCC bugs. Details at: `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=`. Bug ids are sorted by date the bug was confirmed as new. 82

7.1 Peace of Notepad code. 92
7.2 An example of a problem extracted from the documentation of optimization module of GCC. This example can be find at: `http://www.cin.ufpe.br/~sfs/survey/first.php?question=1&user=10`. 93

List of Tables

3.1 Subject SPLs. 46
3.2 Experimental Results for Various Techniques 49

Contents

1 Introduction **17**

1.1 Problem Overview . . . 19

1.1.1 High Dimensionality . . . 20

1.1.2 Lack of Feature Models . . . 21

1.2 Solution Overview . . . 22

1.2.1 High Dimensionality . . . 22

1.2.2 Lack of Feature Models . . . 22

1.2.3 `SPLat` and `SPLif` . . . 23

1.3 Contributions . . . 23

1.4 Collaboration . . . 24

1.5 Outline . . . 24

2 Background **26**

2.1 Feature Models . . . 26

2.1.1 Graphical Representation . . . 27

2.1.2 Propositional Representation . . . 28

2.1.2.1 Modeling Incomplete Feature Models . . . 28

2.2 Building Approaches . . . 29

2.2.1 Compositional . . . 29

2.2.2 Annotative . . . 30

2.2.3 Variability Encoding . . . 30

2.2.3.1 Translating an SPL code from Annotative to Variability Encoding Format . . . 31

2.3 Product Line Testing . . . 33

2.3.1 Case: Testing GCC . . . 34

3 SPLat **36**

3.1 Example . . . 36

3.2 Technique . . . 38

3.2.1 Feature Model Interface . . . 38

3.2.2 Main Algorithm . . . 39

3.2.3 Example Run . . . 41

3.2.4 Reset Function . . . 42

3.2.5 Potential Optimization . . . 43

3.2.5.1 Feature Expression . . . 43

3.2.6 Implementation . . . 44
3.3 Evaluation . . . 44
3.3.1 Subjects . . . 44
3.3.2 Tests . . . 46
3.3.3 Comparison Techniques . . . 48
3.3.4 Results . . . 48
3.3.5 Case Study: Groupon . . . 52
3.3.5.1 SPLat Application . . . 52
3.3.5.2 Results . . . 52
3.3.6 Threats to Validity . . . 54

4 SPLif **55**
4.1 Illustrative Example . . . 55
4.1.1 Notepad . . . 55
4.1.2 A Notepad test (failure) . . . 55
4.1.3 SPLat . . . 56
4.1.4 SPLif in a nutshell . . . 56
4.1.5 SPLif on Notepad Tests . . . 57
4.2 Technique . . . 57
4.2.1 Test Exploration . . . 58
4.2.2 Test Ranking . . . 59
4.2.3 Configuration Ranking . . . 60
4.2.4 Algorithm . . . 60
4.3 Evaluation . . . 62
4.3.1 Subjects . . . 62
4.3.2 Setup . . . 63
4.3.2.1 Tests analyzed . . . 63
4.3.2.2 Initial Feature Model and Ground Truth . . . 65
4.3.3 Ranking Tests Using Suspiciousness Score . . . 65
4.3.4 Ranking Configurations . . . 66
4.3.4.1 Discussion . . . 68
4.3.5 Incremental Runs of SPLif . . . 69
4.3.6 Discussion of Test Failures . . . 70
4.3.6.1 Companies . . . 70
4.3.6.2 DesktopSearcher . . . 70
4.3.6.3 GPL . . . 71
4.3.6.4 Notepad . . . 72
4.3.6.5 ZipMe . . . 72
4.3.7 Threats to Validity . . . 73

5 Case Study: GCC **74**
5.1 Research Questions . . . 74
5.2 General Infrastructure . . . 75
5.2.1 The GCC Testing Infrastructure . . . 75
5.2.2 Implementation . . . 75
5.2.2.1 Instrumentation . . . 75
5.2.2.2 Execution . . . 76
5.3 Setup . . . 78
5.3.1 Tests Execution . . . 78
5.3.2 Tests Analyzed . . . 78
5.3.3 Options Analyzed . . . 78
5.3.4 Initial Feature Model and Ground Truth . . . 78
5.4 Results . . . 79
5.4.1 Ranking Tests and Configurations . . . 79
5.4.2 New Bugs Found . . . 81
5.4.3 New Configuration Constraints Found . . . 82
5.5 Threats to Validity . . . 83

6 Related Work **85**
6.1 SPLat . . . 85
6.1.1 Dynamic Analysis . . . 85
6.1.2 Static Analysis . . . 86
6.2 SPLif . . . 87
6.2.1 Product Line Testing . . . 87
6.2.2 Feature Model Extraction and Inference . . . 88
6.2.3 Fault Localization . . . 89
6.2.4 Configuration Troubleshooting . . . 89

7 Conclusion and Future Work **91**
7.1 Future Work . . . 91
7.1.1 Feature Model Inference . . . 92
7.1.1.1 Feature Model Inference from Code - SPLand . . . 92
7.1.1.2 Feature Model Inference from Documentation - MIHCO . . . 93
7.1.2 Regression Testing of Software Product Lines . . . 94

References **96**

1 Introduction

In general, software engineering provides two traditional approaches to develop systems focusing on customer needs: (1) including all possible functionalities that a customer might ever need, or (2) producing a single purpose system by order of a single customer. The former approach may be very general for the customers needs, in the latter there is a considerable cost related to producing a tailor-made solution. For instance, in the development perspective, having n distinct projects for developing n distinct software products may lead to both decreased productivity and increased time-to-market. In addition, requirements are growing not only in number, but also in complexity, to satisfy as many customers as possible in as little time as possible.

Instead of having one project for each software product, one possible way to solve all those observed problems would be having only one project containing all products. The idea is to first focus on developing artifacts that are common to all products, and next, focusing on developing different artifacts to satisfy specific customer requirements, represented by *features* (increments in functionality). Those different features can be combined, resulting in different products.

Configurable Systems (CSs) and *Software Product Lines (SPLs)*[1] are systems that typically adopt those practices during their development. They can be defined as "*a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way*" (CLEMENTS; NORTHROP, 2001). Core assets correspond to reusable artifacts or resources that are used to specify or build different products[2] that can be configured through selection of features (or configuration variables).

In practice, SPLs or CSs can be adapted according to a set of features, represented by configuration variables at the code level. For instance, the web browser Firefox, the GNU Compiler Collection (GCC), and the Linux Kernel are examples of systems with such characteristics.

From market perspective, SPLs have been proposed as a way to increase productivity

[1] *Configurable Systems* and *Software Product Lines* are used as synonym in this work.

[2] For ease of exposition, we use the terms "program", "product", "combination of features" and "configurations" interchangeably.

in developing highly-configurable software systems (POHL; BÖCKLE; LINDEN, 2005). The underlying principle is that they can systematically reuse software artifacts, instead of developing software systems from scratch, thus reducing time-to-market in an industrial scenario where customers demand a different set of features for one existing application. Cases where the use of SPLs has demonstrated benefits include operating systems, telecommunication software, and mobile apps (BERGER et al., 2010; ZAVE, 1993; AHO; GRIFFETH, 1995; ALVES et al., 2005; YOUNG, 2005).

Software Product Line Engineering (SPLE) is a methodology for developing families of products through reuse of a common (commonality) and variable (variability) set of assets (POHL; BÖCKLE; LINDEN, 2005). While commonality is represented by the common assets found in all products, variability is expressed by features, that are included or excluded from individual products. At the application level, variability is key to increase both flexibility and reuse during development, by reducing software engineering costs. However, it moves those costs to testing activities, by posing many challenges for testing SPLs.

Software testing is a popular approach used in industry to assure software quality. Specifically, testing software means checking if the observed system behavior matches the specified, through a controlled execution. The main objective of this task is to find faults before they manifest as failures. In the context of SPLs, testing activities should consider, among other practical considerations, the SPL variability (POHL; BÖCKLE; LINDEN, 2005), that represents a great challenge for testing SPLs, in contrast to testing a single-system developed in a traditional way.

One of the challenges for testing SPLs is the variability space that grows exponentially with the number of features. For instance, if an SPL has n boolean features, there are 2^n possible feature combinations. Therefore, testing this kind of system is expensive as it may require running each test against all those combinations. We call this problem the *High Dimensionality Problem*.

Testing an SPL becomes further complicated by the unavailability or incompleteness of the *Feature Model (FM)*. An FM models variability, by documenting features and their relationships, and also defines legal feature combinations or configurations, *i.e.*, it can detect inconsistent (illegal) configurations. For example, if a test fails in a scenario without a complete FM, it is difficult to determine if there is a fault in the code or in the test, instead of a failure due to inconsistent combinations of features. For this reason, the lack or incompleteness of FM presents a great challenge to discover the true causes for test failures. We call this problem the *Lack of Feature Model Problem*.

Considering both the High Dimensionality Problem and the Lack of Feature Model Problem, we define the scope, assumption, and hypothesis of this thesis as follows:

- **Scope.** The scope of this work comprises software product lines and configurable systems that build products through dynamically bound features, known as *Variability Encoding c.f.*, Section 2.2.3. In this methodology, features are represented as boolean

variables (*feature variables*) that guard portions of code that belong to them. These feature variables are enabled and disabled during code execution. Figure 1.1 illustrates a piece of code developed using this methodology, where T, T, and M are feature variables.

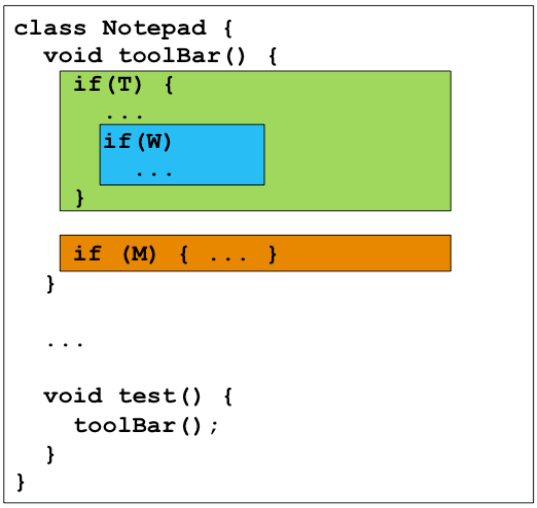

Figure 1.1: Example of code using dynamically bound features.

In theory, the proposed solutions can be applied to all approaches to build SPLs. However in practice, the implemented solutions should be adapted to support other kinds of SPLs' implementation. The systems `groupon.com` website and GCC (GCC - Test, 2014; GARVIN; COHEN, 2011) are developed according to this methodology;

- **Assumption.** We assume that tests can reveal errors when executed on multiple configurations, even when they were developed for one specific configuration. The tests for `groupon.com` website (KIM et al., 2013), GCC (GCC - Test, 2014; GARVIN; COHEN, 2011) and Firefox (GARVIN; COHEN, 2011) are developed in this manner. Figure 1.2 presents an example of GCC test designed for an specific configuration, but, in practice, the GCC test infrastructure supports running this test against several configurations, since they respect the features $-O3$ and $-fcompare-debug$;

- **Hypothesis.** Our hypothesis is that it is feasible to test SPLs on multiple configurations in a systematic way at a low cost, and find errors even without documented feature model.

1.1 Problem Overview

In this section, we detail both the High Dimensionality and the Lack of Feature Models problems.

```
1  /* PR debug/47684 */
2  /* {dg-do compile} */
3  /* {dg-options "-O3 -fcompare-debug"} */
4  /* {dg-xfail-if "" {powerpc-ibm-aix*} {"*"} {""} } */
5
6  int in[8][4];
7  int out[4];
8
9  void
10 foo (void)
11 {
12   int sum = 1;
13   int i, j, k;
14   for (k = 0; k < 4; k++)
15     {
16       for (j = 0; j < 4; j++)
17         for (i = 0; i < 4; i++)
18           sum *= in[i + k][j];
19       out[k] = sum;
20     }
21 }
```

Figure 1.2: Example of test that was designed for an specific configuration: $-O3$ and $-fcompare-debug$.

1.1.1 High Dimensionality

Systematically testing configurable systems and SPLs is difficult because running each test can, in principle, require many actual executions—one execution for each possible configuration or feature combination. Thus, one test does not simply encode one execution of a program; the cost of running a test suite is proportional to the number of tests times the number of configurations.

Current techniques for handling this combinatorial problem can be divided into sampling and exhaustive exploration. Sampling uses a random or sophisticated selection of configurations, *e.g.*, pair-wise coverage (COHEN; DWYER; SHI, 2007). However, such selections can run a test on several configurations for which the test executions are provably equivalent (thus only increasing the test time without increasing the chance to find bugs), or it can fail to examine configurations that can expose bugs (APEL et al., 2013). Exhaustive exploration techniques consider all configurations, but can use optimization approaches to prune redundant configurations that need not be explored (APEL et al., 2013; D'AMORIM; LAUTERBURG; MARINOV, 2007; KäSTNER et al., 2012; KIM; BATORY; KHURSHID, 2011; KIM; KHURSHID; BATORY, 2012; RHEIN; APEL; RAIMONDI, 2011). Such works use either static analysis (KIM; BATORY; KHURSHID, 2011) or heavyweight dynamic analysis based on model checking (APEL et al., 2013; D'AMORIM; LAUTERBURG; MARINOV, 2007; KäSTNER et al., 2012; KIM; KHURSHID; BATORY, 2012; RHEIN; APEL; RAIMONDI, 2011). These techniques are safe, *i.e.*, they do not miss bugs. However, they may be heavy and, consequently, slow.

1.1.2 Lack of Feature Models

Feature models formally capture dependencies among features. They distinguish which combinations of features are consistent from those that are not. If an SPL has n boolean features, there are 2^n possible combinations, called configurations. If the feature model can classify each of the 2^n configurations as consistent or inconsistent[3], we consider this model as *complete*. On the other hand, if the feature model cannot classify all of the 2^n configurations as consistent or inconsistent, this model is *incomplete*, *i.e.*, it does not have the needed knowledge to distinguish all combinations of features are consistent from those that are not. In the limit, this incomplete model can be empty, *i.e.*, it cannot classify any configuration (*c.f.*, Section 2.1).

A relevant problem for testing SPLs (KIM et al., 2013; BORBA et al., 2013; APEL et al., 2013; SHI; COHEN; DWYER, 2012; SONG; PORTER; FOSTER, 2012; KIM; KHURSHID; BATORY, 2012; KIM; BATORY; KHURSHID, 2011; GARVIN; COHEN, 2011; CABRAL; COHEN; ROTHERMEL, 2010) is to reduce the number of tests to run. In this context, feature models play a key role by constraining the space of products to test and enables accurate categorization of failing tests as failures of programs or the tests themselves, not as failures due to inconsistent configurations. However, *most* cited prior work on testing SPLs assumes the availability of a complete formally-specified feature model.

Unfortunately, in practice, feature models are not always available. Indeed, this absence presents a great challenge for testing an SPL. For instance, during `SPLat` experiments we realized that many tests for the codebase of the `groupon.com` website (KIM et al., 2013) failed, and it was not possible distinguish if the failures were due to inconsistent configurations or due to a bug in code or tests, because there is no formally-specified feature model.

There is a large body of work that mitigates that problem by inferring/extracting feature models as complete as possible (CZARNECKI; WASOWSKI, 2007; ALVES et al., 2008; WESTON; CHITCHYAN; RASHID, 2009; SHE et al., 2011; RABKIN; KATZ, 2011; ANDERSEN et al., 2012; LOPEZ-HERREJON et al., 2012; ACHER et al., 2012; HASLINGER; LOPEZ-HERREJON; EGYED, 2013; DAVRIL et al., 2013; XU et al., 2013) that include: static analysis to extract feature dependencies from code, information retrieval and data mining, evolutionary search, and algorithms based on propositional logic. These approaches face some problems that we can avoid, because it is not necessary to discover a whole feature model in order support the testing of SPLs.

Our insight is to discover only the relevant part of the FM to check the consistency of faulty revealing configurations. Thus, we can avoid some problems that approaches for inferring/extracting feature models have to deal, for example: (1) the key constraints that comprise a feature model need to be mined; (2) the relationships need to be arranged in a manner that leads to an intuitive and understandable model; (3) a feature may have a range of values,

[3] An inconsistent configuration is a configuration that violates some rule of the FM. Otherwise, it is a consistent configuration.

instead of being boolean; (4) the inferred/extracted information is harvested from a variety of source code artifacts, makefiles, comments, and documentation.

1.2 Solution Overview

In this work, we propose two solutions to addresses both the High Dimensionality and the Lack of Feature Models problems.

1.2.1 High Dimensionality

In order to solve the aforementioned problems related to the High Dimensionality problem, we present `SPLat`, a new lightweight technique to reduce the cost of systematically testing SPLs and highly configurable systems. Our intuition is that the configurations to run against a test can be determined *during* test execution, rather than using an up-front static analysis. `SPLat` minimizes the number of configurations for the tests. Experimental results show that `SPLat` yields a reduction in testing time proportional to the reduction in the number of configurations. Our insight into pruning configurations was inspired by the Korat algorithm for test-input generation (BOYAPATI; KHURSHID; MARINOV, 2002), which introduced the idea of *execution-driven pruning* for solving data-structure invariants written as imperative code.

Initially, `SPLat` requires a feature model[4] to make it supporting constraints among configuration variables, which delimit the space of configurations to explore. For an SPL, these constraints are expressed through a feature model (KANG et al., 1990) that (1) provides a hierarchical arrangement of features and (2) defines allowed configurations. `SPLat` uses SAT to prune *inconsistent* configurations and in tandem uses execution-driven pruning to further remove unnecessary consistent configurations for execution of each test. Moreover, `SPLat` is effective because it monitors the accesses of configuration variables during test execution. Monitoring is lightweight in terms of both its execution overhead and its implementation effort.

1.2.2 Lack of Feature Models

In order to solve this problem, we propose `SPLif`, a technique for testing SPLs that does not require a priori availability of complete feature models. `SPLif` does not require the entire model in order to validate fault-revealing configurations. Instead, `SPLif` guides the developer in incrementally classifying configurations related to the parts of the model that are relevant for failing test runs. Our insight is that by running each test against many configurations, we can utilize information from failing and passing runs to help developers prioritize their inspection of failures.

[4] `SPLat` only worked with an incomplete or absent feature model after `SPLif`, for more detail see Section 4.2.1.

1.2.3 SPLat and SPLif

The connection between SPLat and SPLif addresses both the High Dimensionality and the Lack of Feature Models problems. Figure 1.3 provides an overview of our solution, whose inputs are: the system source code, its tests, and optionally a feature model that can be complete, incomplete, or empty[5]. From these inputs, SPLat executes the tests by exploring only configurations that are touched during test execution. If the FM is not empty, SPLat prunes inconsistent configurations during the exploration. As a result SPLat produces a profile of passing and failing tests and respective explored configurations.

From that profile, SPLif produces a ranking of failing tests and configurations that are indicative of a problem (in test or code). These failing configurations are inspected by the user, with prior knowledge of the SPL being tested to classify it as consistent or inconsistent. If the configuration is inconsistent, then it improves the existing incomplete FM or incrementally builds an FM, if it is empty. If the configuration is consistent, it indicates a real problem in the code or test, and the user can make a repair in the code or test.

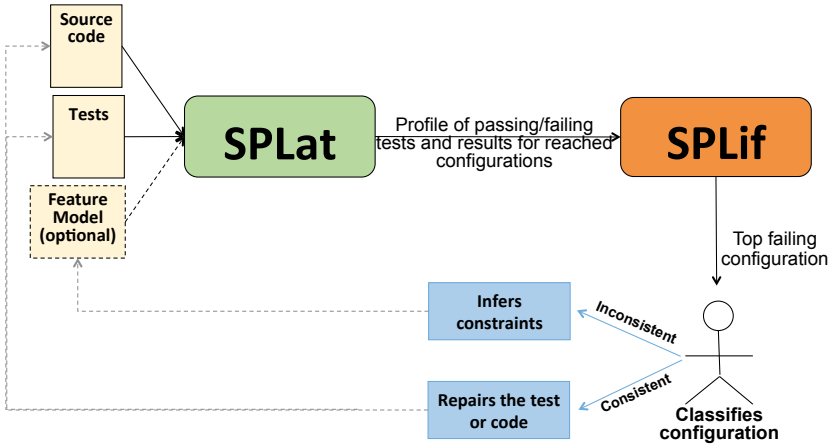

Figure 1.3: Overview of the solution.

1.3 Contributions

This work makes the following contributions:

- **Approach.** We introduce the idea of lightweight monitoring for highly configurable systems to speed up test execution. SPLat instantiates this idea and can be easily implemented in different run-time environments. We also present SPLif, a technique that synergistically exploits tests and incomplete feature models (in the limit, starting

[5]Since we are presenting an overview of the solutions together, we are assuming that the FM may be optional, but Chapter Section 3 considers a complete FM, except by Groupon, where SPLat ignored the FM exploration, because this system does not have an FM. Chapter Section 4 considers the FM as optional, and the final solution works in this manner.

from an empty feature model) to help users both (1) distinguish test failures caused by problems in test or code from those caused by inconsistent configurations and (2) build a more complete feature model, as a consequence.

- **Implementation.** It was developed three versions of `SPLat` (one for Java, one for Ruby on Rails[6], and one for C) for exploring SPL tests on various reachable configurations. And we developed `SPLif` (SOUTO et al., 2015), that builds on the `SPLat` tool (KIM et al., 2013), to speed up discovery of bugs in SPLs and CSs that do not require a priori availability of feature models.

- **Evaluation.** We evaluate `SPLat` on 10 Java SPLs. Experimental results show that `SPLat` effectively identifies relevant configurations with a low overhead. We also apply `SPLat` on a large configurable system (with over 171KLOC in Ruby on Rails). The system uses over 170 configuration variables and contains over 19K tests (with over 231KLOC in Ruby on Rails), and `SPLat` scales to this size without much engineering effort. `SPLif` has been evaluated on five SPLs. The results demonstrate the utility of `SPLif` in automating testing of SPLs with incomplete feature models. Furthermore, we evaluated `SPLif` on one large extensively tested configurable system, GCC, where it helped to reveal 5 new bugs by running existing tests with different configurations and inspecting failures in an order that allows to quickly find the bugs, 3 of which have been fixed after our bug reports.

1.4 Collaboration

`SPLat` was developed in an independent manner and resulted in a publication together with Chang Hwan Peter Kim, Don Batory and Sarfraz Khurshid from the University of Texas, and Darko Marinov from the University of Illinois (USA). We joined our works (implementations and experiments) and we had a camera ready version at the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2013 (KIM et al., 2013).

`SPLif` was designed, implemented, and evaluated in collaboration with Darko Marinov from the University of Illinois (USA), Divya Gopinath, Sarfraz Khurshid and Don Batory from the University of Texas at Austin (USA). Together, we published a paper in the ACM International Conference on Software Product Lines, SPLC 2015 (SOUTO et al., 2015).

1.5 Outline

We organize this document as follows:

[6]The Ruby version of `SPLat` was developed and evaluated by professor Darko Marinov from the University of Illinois (USA).

- Chapter *2* reviews essential concepts used throughout this work;
- Chapter *3* presents `SPLat`, our approach to address high dimensionality. Firstly, we discuss the general idea through an illustrative example. Then, we explain the technique and its implementation. Finally, we detail the evaluation and discuss the results;
- Chapter *4* presents `SPLif`, our approach to deal with the lack of feature model. Firstly, we present some specific definitions, and discuss the general idea through an illustrative example. Then, we explain the technique and its implementation. Finally, we detail the evaluation, discuss the results and limitations;
- Chapter *5* presents a case study conduced with the aim of demonstrate `SPLat` and `SPLif` with the GNU Compiler Collection (GCC);
- Chapter *6* discusses related work;
- Chapter *7* presents the final considerations of this work, and proposes future research based on this work.

2

Background

In this chapter we introduce essential concepts and terminology to support the discussion of the remaining chapters. Firstly, we introduce some concepts related to *Software Product Lines* in Section 2.1. In this context, we highlight the key elements used in this work: *features*, *configurations*, and *feature models*. Then, in Section 2.2, we discuss the main approaches used to build products[1]. Finally, in Section 2.3, we present the product line testing approach used in this work.

SPLs have been proposed as a way to improve both the efficiency of the software development process and the quality of the software developed, by allowing engineers to systematically build families of products through the reuse of a common (commonality) and variable (variability) set of *core assets* (CLEMENTS; NORTHROP, 2001; POHL; BÖCKLE; LINDEN, 2005). While commonality is represented by the common assets found in all products, variability is expressed by features, that are included or excluded from individual products.

Core assets correspond to reusable artifacts or resources that are used to specify or build different products. A feature characterizes one distinct software capability. For example, the capability of a game for rendering figures in 3D (ALVES et al., 2005). For our purposes, a feature can simply be seen as a variable or an option whose value influences code selection. A product (or configuration) is an n-digit boolean $f1...fn$ representing feature assignments for a product line with n features.

2.1 Feature Models

In order to obtain the benefits of the SPL approach, it is necessary to model variability. And there are a number of different techniques for modeling variability (KANG et al., 1990; CZARNECKI; HELSEN; EISENECKER, 2005; BATORY, 2005; SCHOBBENS et al., 2007; GHEYI; MASSONI; BORBA, 2008; WEISS et al., 2008; SCHMID; RABISER; GRüNBACHER, 2011), according to different focus and goals. The most popular form to model variability is

[1] For ease of exposition, we use the terms “program”, “product”, “combination of features” and “configurations” interchangeably.

through *feature models (FMs)* and their graphical representation as *feature diagrams*. A feature model documents the features of a product line and their relationships (APEL et al., 2013), and also defines legal feature combinations or configurations. We focus on two different representations for expressing FMs: (1) graphically as a tree, and (2) as propositional formulas.

2.1.1 Graphical Representation

A feature model can be graphically represented as a tree, where nodes correspond to features and edges correspond to feature dependencies. The root of the tree often identifies the subject application and other nodes describe features. Edges define feature hierarchy: if a node i is an ancestral of j in the tree then any product that contains feature j also contains i. Edges of the tree are decorated to indicate special kinds of relationships across features, see Figure 2.1. The options of decoration are as follows:

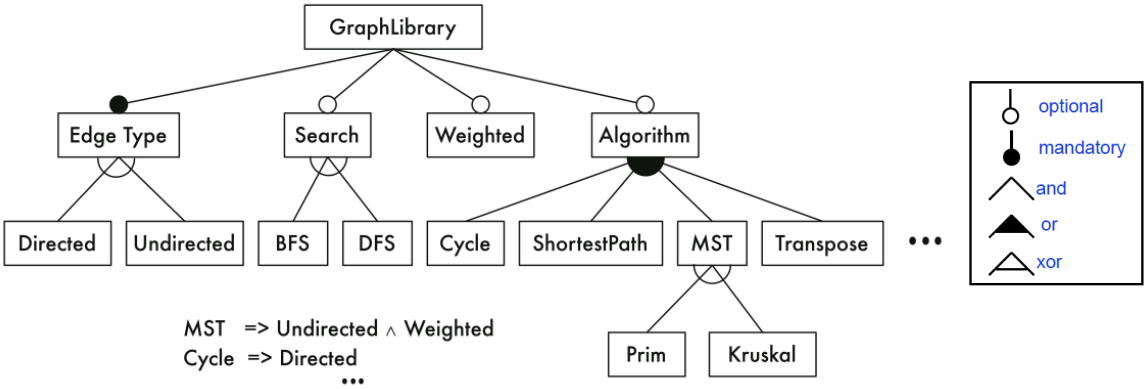

Figure 2.1: A possible feature diagram of the Graph Library SPL. Source (APEL et al., 2013).

- **Filled circle (mandatory)**: An edge from feature A to feature B that terminates in a filled circle denotes a *mandatory* feature. Feature B must be selected if A is selected.
- **Empty circle (optional)**: Feature B may or may not be selected if A is selected.
- **Empty arc (xor)**: An empty arc labeling the edges from feature A to features $B1...Bn$ indicate *alternative* (mutually-exclusive) features. Feature Bi can only be selected if Bj is not selected, for $i \neq j$ and $i, j : 1 \leq j \leq n$. Additionally, one feature in the range must be selected if feature A is selected.
- **Filled arc (or)**: A filled arc labeling the edges from feature A to features $B1...Bn$ indicates *or* features. At least one feature in the range must be selected if feature A is selected.
- **No arc (and)**: No arc labeling the edges from feature A to features $B1...Bn$ indicates *and* features. All the features in the range must be selected.

- **Constraints**: The propositional formulas under the tree represents the constraints across features. For example, see the constraints in Figure 2.1.

2.1.2 Propositional Representation

Let ϕ be a set of boolean variables denoting SPL features. A *configuration* $c : \phi \rightharpoonup \{false, true\}$ is a partial function from variables to boolean values; c maps *some* (not necessarily all) feature variables to values *false* or *true*. A configuration can be encoded as a boolean formula $f_c = \bigwedge p_i$, where $p_i = (x_i | \neg x_i)$ for $x_i \in \phi$. We denote with $|f_c|$ the number of variables referenced in f_c. We say that a configuration c is *complete* iff $|f_c|{=}|\phi|$; it is *incomplete* otherwise. An incomplete configuration c represents a set of $2^{|\phi|-|f_c|}$ complete configurations (we call them *extensions* of f_c), which map *all* variables to boolean values.

Example. Let $\phi = \{A, B, C, D, E\}$. Configuration $f_{c_1}{=}A \wedge B \wedge \neg C \wedge D \wedge E$ is complete. Configurations $f_{c_2}{=}A \wedge \neg B$ and $f_{c_3}{=}A \wedge B \wedge \neg C \wedge E$ are incomplete. c_2 can be also written as "10???", where we assume a linear order over ϕ: $A < \ldots < E$ and use the numbers 1 and 0 to indicate, respectively, the presence or absence of a feature and the symbol "?" to indicate "undefined". It is important to note that "10***" *extensions* are: "10000", "10001", "10010", "10100", "10101", "10110", "10011", "10111".

A feature model defines which configurations are legal for a given SPL. We use the label "✓" (for *valid*) to indicate that a complete configuration does not violate model constraints, while the label "✗" (for *invalid*) indicates that some constraint has been violated.

2.1.2.1 Modeling Incomplete Feature Models

Assuming that complete knowledge about SPL constraints is **not** available. In that case, an incomplete feature model is a partial function $M\_ : 2^{\phi} \rightharpoonup \{✓, ✗\}$ that maps *complete configuration* $c \in 2^{\phi}$ to a configuration label. If the label of a configuration is not in $M\_$, it is "*unknown*". In one limit, the feature model is *empty* if it assigns an "*unknown*" label to all $2^{|\phi|}$ configurations. In another, the feature model is *complete* if it assigns ✓or ✗ to every configuration.

Analogous to the labels "✓" and "×" used to indicate validity of complete configurations, we use the labels "*C*" (for *consistent*) and "*I*" (for *inconsistent*) to indicate consistency of *incomplete configurations*. An incomplete configuration $c \in 2^{\phi}$ is inconsistent if it violates some constraint in the feature model; it is consistent otherwise. We model an incomplete feature model as a total function $M : 2^{\phi} \rightarrow \{C, I\}$.

Example. Formula $f_M{=}(x \rightarrow \neg z) \wedge (\neg y \rightarrow z)$ encodes an incomplete feature model M. The configuration 1?1?? is inconsistent as it violates $x \rightarrow \neg z$ whereas the configuration 11?11 is consistent as it does not violate any constraint in M. Indeed, it is possible to obtain consistent complete configuration (aka valid ✓) in M from 11?11 with the assignment of 0 to z, however the assignment of 1 to z produces an inconsistent complete configuration (aka invalid ✗).

Note that similarity to consistent configurations does not imply consistency: when an incomplete configuration is consistent, extensions of that configuration *may or may not* be consistent. In contrast, when an incomplete configuration is inconsistent, all its extensions *must be* inconsistent. Provided that one preserves this invariant, it is possible to augment f_M incrementally by reducing uncertainty associated with incomplete configurations.

2.2 Building Approaches

The *build* process of an SPL takes a selection of feature requests as input and outputs a *product* that implements those features. Products share a common codebase and vary in the set of features they implement. There are many different approaches to build products, such as *Compositional*, *Annotative*, and *Variability Encoding*. In the following we detail each of these approaches.

2.2.1 Compositional

Feature BASE

```
1  class Stack{
2    void push(Object o) {
3      elementData[size++] = 0;
4    }
5  }
```

Feature LOCKING

```
6  refines class Stack{
7    void push(Object o) {
8      Lock l = lock(o);
9      Super.push(o);
10     l.unlock();
11   }
12   Lock lock() {/*...*/}
13 }
14 class Lock {/*...*/}
```

Feature LOGGING

```
14 refines class Stack{
15   void push(Object o) {
16     Super.push(o);
17     log("added " + o);
19   }
20   void log(String msg) {/*...*/}
21 }
```

(a) Implementation with Jak.

Feature BASE

```
1  class Stack{
2    void push(Object o) {
3      elementData[size++] = 0;
4    }
5  }
```

Feature LOCKING

```
6  aspect Locking{
7    around(Object o, Stack stack):
8      execution(void Stack.pusk(..))
9      && args(o) && this(stack)
10   {
11     Lock l = lock(o);
12     proceed(o);
13     l.unlock();
14   }
15   Lock Stack.lock(Object o) { ... }
16   static class Lock {/*...*/}
17 }
```

Feature LOGGING

```
18 aspect Logging{
19   after(Object o):
20     execution(void Stack.push(..))
21     && args(o)
22   {
23    log("added " + o);
24   }
25   void log(String msg) {/*...*/}
26 }
```

(b) Implementation with AspectJ.

Figure 2.2: Two compositional implementations of a stack example with three features. Source (KASTNER., 2010).

In compositional approaches features are implemented separately in distinct modules (files, classes, packages, plug-ins, etc.). These modules can be composed in different combinations in order to build products. The key challenge of composition-based approaches is to keep the mapping between features and modules simple and tractable (KASTNER., 2010; APEL

et al., 2013). Figure 2.2 shows an example of two compositional implementations of a simple stack example that can be extended by two features *LOCKING* and *LOGGING*. The example in Figure Figure 2.2a uses class refinements of the feature-oriented language Jak (BATORY; SARVELA; RAUSCHMAYER, 2003). Figure Figure 2.2b represents the same application by using the aspect oriented language AspectJ (KICZALES et al., 2001) to implement the same features.

2.2.2 Annotative

A very popular (and old) implementation mechanism of an SPL appears in programs that combine make files and *conditional compilation*, as provided by the C compiler pre-processor (CZARNECKI; EISENECKER, 2000). In this approach, the user annotates the code with `#ifdef`-like directives guarded by symbols that denote features and use `make` files to generate a product with one selection of features enabled. Figure 2.3 illustrates this kind of code representation. Annotative approaches are widely used in practice because they are easy to use. Nevertheless, they are often criticized for a number of problems, such as lack of modularity, reduced readability, and being error-prone (KASTNER., 2010; APEL et al., 2013).

```
1  class Node {
2          int id = 0;
3
4          //#ifdef NAME
5          private String name;
6          String getName() { return name; }
7          //#endif
8          //#ifdef NONAME
9          String getName() { return String.valueOf(id); }
10         //#endif
11
12         //#ifdef COLOR
13         Color color = new Color();
14         //#endif
15
16         void print(){
17                 //#if defined(COLOR) && defined(NAME)
18                 Color.setDisplayColor(color);
19                 //#endif
20                 System.out.print(getName());
21         }
22 //#ifdef COLOR
23 Class Color {
24         static void setDisplayColor(Color c) {*...*}
25 }
26 //#endif
27 }
```

Figure 2.3: Annotative implementation using conditional compilation for Graph Library SPL. Source (APEL et al., 2013).

2.2.3 Variability Encoding

The Variability Encoding representation (POST; SINZ, 2008; APEL et al., 2013) delays the binding time of code variations to execution time. Similar representations have been used

to assist automated code analysis of configurable systems (REISNER et al., 2010; APEL et al., 2011; KäSTNER et al., 2012; KIM; KHURSHID; BATORY, 2012). More specifically, for each feature, there is a *feature variable*, which is a static boolean field whose value, for one configuration, is determined at the beginning of program execution and must remain fixed throughout the execution. Figure 2.4 shows a possible variability encoding for the graph example in Figure 2.3. The presence and absence of the features *NAME*, *NONAME*, and *COLORED* are modeled by three corresponding Boolean variables, located in class *Conf*. Code that is specific to particular features is executed conditionally based on the values of these variables (highlighted in Figure 2.3) (APEL et al., 2013).

```
1  class Node {
2          int id = 0;
3          private String name;
4          String getName() {
5                  if (Conf.NAME) return name;
6                  if (Conf.NONAME) return String.valueOf(id);
7                  throw VariabilityException();
8          }
9          Color color = new Color();
10
11         void print(){
12                 if (Conf.COLOR && Conf.NAME)
13                         Color.setDisplayColor(color);
14                 System.out.print(getName());
15         }
16 }
17
18 Class Color {
19         static void setDisplayColor(Color c) {*...*}
20 }
```

Figure 2.4: Variability Encoding implementation for Graph Library SPL, conditionally executed code is highlighted. Source (APEL et al., 2013).

Although the Annotative approach is widely used in practice and the Compositional one is well-known, we decided to develop `SPLat` and `SPLif` to support SPLs implemented with Variability Encoding approach, because it is more appropriate and practical for the types of code analysis we need to do, *i.e.*, each feature is modeled as a code variable and it allows checking their states at execution time. Moreover, the real cases we studied, Groupon and GCC, adopt this building approach. We could have been used Annotative or Compositional approaches with some engineering effort would be needed, the next section details this effort.

2.2.3.1 Translating an SPL code from Annotative to Variability Encoding Format

Our goal is to translate from a format to the other without loose information. To achieve that, we have to consider two kinds of variations (intra-method and inter-method). A variation is a program member[2] that belongs to a feature. Figure 2.5a illustrates this kind of variation. On the other hand, variation intra-method represents method members that belong to feature(s),

[2]Examples of program member: variables, method declarations, inner classes, attribute declarations (global variables), method parameters, return of methods/functions, or class extension/implementation.

such as variable declarations, method calls. Variation inter-method represents program members out of the method that belong to feature(s), for example: global variables, method parameters, return of methods/functions, or class extension/implementation. Figure 2.6a illustrates this kind of variation.

Our insight to translate from Annotative into Variability Encoding format is to find all the definitions and uses for each variation, and:

- For (1) variations intra-method, we need to guard their definitions and uses. For example, the code from Figure 2.5a is translated into the code of Figure 2.5b. In this case, "*Stack buffer*" belongs to feature "*A*", it means that this variable can only be used when the feature "*A*" is enabled. So, when the code is translated the behavior remains the same;

```
1  class Clazz {
2      void add(Object x){
3      //#ifdef A
4      Stack buffer;
5      //#endif
6      //...
7      //#ifdef A
8      buffer.add(x);
9      //#endif
10     //...
11     }
12 }
```

a A piece of code developed using the Annotative approach with a variation intra-method.

```
1  class Clazz {
2     void add(Object x){
3     if(A)
4      Stack buffer;
5     //...
6     if(A)
7       buffer.add(x);
8     //...
9     }
10 }
```

b A piece of code translated from Annotative (Figure 2.5a) to Variability Encoding approach with a variation intra-method.

Figure 2.5: Variations intra-method.

- For (2) variations inter-method, we need to guard their uses and define default values to avoid code repetition. For example, the code from Figure 2.6a is translated into the code of Figure 2.6b. In this case, the variation in a parameter of the method *m* (at At line 4 from Figure 2.6a) is moved to lines 15-17 from Figure 2.6b. The method declaration have no apparently variation, but its calls have to be preceded by the variation of that parameter. So, we set the variable *x* to its correct value if the feature *A* is enabled. Otherwise, we define a default value. This artifice is used to avoid code repetition, *i.e.*, having two method definitions (one for each variation).

```
1  class Clazz {
2      void m(
3        //#ifdef A int x,//#endif
4        double y){
5        //...
6        //#ifdef A
7        x = 10;
8        //#endif
9        y = 1000;
10       //...
11     }
12 }
```

a A piece of code developed using the Annotative approach with a variation inter-method.

```
1  class Clazz {
2      void m(int x, double y){
3        //...
4        if(A)
5          x = 10;
6        y = 1000;
7        //...
8      }
9  }
10
11 class ClazzX {
12     Clazz c = new Clazz();
13     void mx(){
14      int x;
15      if(A)
16       x = 10;
17      else x = 0; //a default value
18      c.m(x, 5.0);
19     }
20 }
```

b A piece of code translated from Annotative (Figure 2.6a) to Variability Encoding approach with a variation inter-method.

Figure 2.6: Variations inter-method.

However, even translating the SPL from the Annotative or the Compositional format to the Variability Encoding format, we may have some cases where the variation is in the type of an attribute or in the list of implements or extends. In these cases it would be necessary to have more than one version of the code to support those variations, that means more combinations to run. This may affect the performance of `SPLat`, and it may lead a combinatorial explosion, in the worst case.

2.3 Product Line Testing

A *test* of a product line is simply a test that executes some methods, references some code of the product line and produces a result that can be checked against the expected result. Figure 2.7b shows a sample test that calls the method in Figure 2.7a. It is very important to note that a feature variable can have different values depending on which configuration is being executed for a test. Without analyzing the feature model or the code, the test must be executed for every configuration of the feature model, i.e. for each configuration, feature variables' values

must be set using the configuration's feature assignments and the test must be executed.

```
1  class Clazz{
2    int value;
3    //…
4    int getValue() {
5     //…
6     if(A) {
7      value = 1;
8     }
9     //…
10    if(B) {
11     value = 2;
12    }
13    //…
14    return value;
15   }
16 }
```

(a) Sample SPL code using variability encoding.

```
1  Class Test{
2    //…
3    @Test
4    void test() {
5     Variables.setA(true);
6     Clazz clazz = new Clazz();
7     assertEquals(1, clazz.getValue());
8    }
9  }
```

(b) Sample SPL Test.

Figure 2.7: Sample SPL Code and Test.

When necessary, the terms *test suite*, *test*, and *test case* are distinguished. A *test suite* is a collection of tests. A *test* or a *test case* can have values assigned to feature variables.

2.3.1 Case: Testing GCC

In order to illustrate how a test of an industrial case is designed, we present GCC and then we discuss a test example.

GCC (2014) is a compilation framework with front-ends for a variety of languages and back-ends for a variety of platforms. We are using GCC version 4.8.2 that has 2.015 features, also called configuration options (GCC Options, 2014). GCC has 164 test suites, where 15 are specific for the C language, and the rest is related to other front ends. All test suites comprise 17,492 files of tests in total, each test containing an arbitrary number of assertions.

```
1  /*{ dg-do compile }*/
2  /*{ dg-options "-std=gnu89 -Wformat"}*/
3
4  #include format.h
5
6  void
7  foo (char **sp, wchar_t **lsp)
8
9     /*%a formats for allocation, only recognized in C90 mode, are a
10      GNU extension. Followed by other characters, %a is not treated
11      specially.
12
13    scanf("%as", sp);
14    scanf("%aS", lsp);
15    scanf("%a[bcd]", sp);
16 }
```

Figure 2.8: An example test ext-4.c from GCC test suite.

The test suite from GCC uses DejaGnu (DejaGnu, 2014) as a framework for testing. DejaGnu provides directives (DejaGnu directives, 2014) to define:

- *Actions*: specify the type of a test: preprocess, compile, assemble, link, or run. See Figure 2.8 line 1;

- *Options*: specify some specific configuration variables to run on a test, see Figure 2.8 line 2;
- *Outcomes*: specify the possible outputs: fail, pass, warning, error, etc.

By default, tests from GCC can run on several configurations. However, the test code must define, at the beginning of the test code, feature variables and values (*options* (GCC Options, 2014)) that must be respected by the configurations that will run with such test. For example, dg-options "-std=gnu89 -Wformat", at line 4 in Figure 2.8, determine that configurations for this test must contain the two options: "-std=gnu89" and "-Wformat". The configuration to run on this test must follow this rule, otherwise, the test shall fail for this reason, *i.e.*, the test must run with any configuration that contains both these two options and any other option.

3

SPLat

Many programs can be configured through dynamic and/or static selection of configuration variables. A *software product line*, for example, specifies a family of programs where each program is defined by a unique combination of features. Systematically testing SPL programs is expensive as it may require running each test against a combinatorial number of configurations. Fortunately, a test is often independent of many configuration variables and needs not to be run against every combination. Configurations that are not required for a test can be pruned from execution.

This chapter presents `SPLat`, a new way to dynamically prune irrelevant configurations: the configurations to run for a test can be determined during test execution by monitoring accesses to configuration variables. `SPLat` efficiently reduces the number of configurations and is lightweight compared to prior works that used static analysis and heavyweight dynamic execution. Experimental results on 10 SPLs written in Java show that `SPLat` substantially reduces the total test execution time in many cases. Moreover, we demonstrate the scalability of `SPLat` by applying it to a large industrial code base.

3.1 Example

In order to illustrate the testing process, we use a simple Notepad product line. Figure 3.1 shows the feature model of Notepad. This model has two mandatory features—the root `NOTEPAD` and `BASE`, and three optional features—`MENUBAR`, `TOOLBAR`, and `WORDCOUNT`. A mandatory feature is always `true`; it is present in every configuration. An optional feature may be set to `true` or `false`. In this example, every Notepad configuration must have a `MENUBAR` or `TOOLBAR`. For example, assigning `false` to both `TOOLBAR` and `MENUBAR` would violate the disjunction constraint and therefore be invalid. In contrast, assigning `false` to one of these two features and `true` to the other feature is valid.

For the SPLs we consider in this work, a feature is a boolean variable within the code. Figure 3.2a shows the code for Notepad. `BASE` (clear color) represents the core functionality, *i.e.*, constructing a `Notepad` with a `JTextArea` that the user types into. `TOOLBAR` (green) adds a

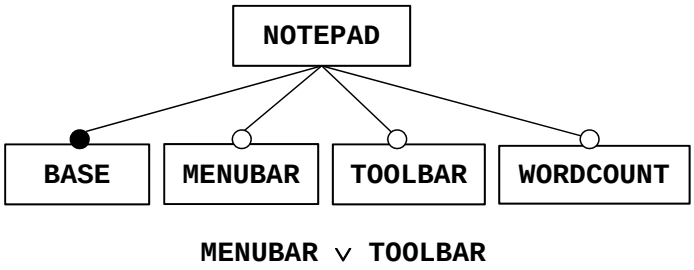

Figure 3.1: Feature Model.

`JToolBar` to the frame. `MENUBAR` (red) sets a `JMenuBar` against the frame. `WORDCOUNT` (blue) adds its toolbar icon if the toolbar is present or its menubar item if the menubar is present.

```
1  class Notepad extends JFrame {
2    Notepad() {
3      getContentPane().add
4                    (new JTextArea());
5    }
6
7    void createToolBar() {
8      if(TOOLBAR) {
9        JToolBar toolBar =
10                     new JToolBar();
11       getContentPane().add
12                 ("North", toolBar);
13
14       if(WORDCOUNT) {
15         JButton button = new
16             JButton("wordcount.gif");
17         toolBar.add(button);
18       }
19     }
20   }
21
22   void createMenuBar() {
23     if(MENUBAR) {
24       JMenuBar menuBar =
25                     new JMenuBar();
26       setJMenuBar(menuBar);
27
28       if(WORDCOUNT) {
29         JMenu menu = new
30                   JMenu("Word Count");
31         menuBar.add(menu);
32       }
33     }
34   }
35 }
```

(a) Code

```
1  void test() {
2   Notepad n = new Notepad();
3   n.createToolBar();
4
5   // Automated GUI testing
6   FrameFixture f = newFixture(n);
7   f.show();
8   String text = "Hello";
9   f.textBox().enterText(text);
10  f.textBox().requireText(text);
11  f.cleanUp();
12 }
```

(b) Test

Figure 3.2: Notepad SPL and Example Test.

Figure 3.2b shows an example test that instantiates the `Notepad` class and creates a toolbar for it. Note that the test does *not* call the `createMenuBar()` method. To be able to execute a test, each variable in the test, except the feature variables, must be given a value.

We use the automated GUI testing framework FEST (FEST, 2014) to run the test. The helper method `newFixture()` is omitted for simplicity. The test execution launches the frame, simulates a user entering some text into the `JTextArea` of the frame, checks that the text area contains exactly what was entered, and closes the frame.

Without analyzing the feature model or the code, this test would need to be run on all 8 combinations of the 3 optional features, to check all potential test outcomes. However, some

configurations need not be run. Analyzing the feature model, we note that two configurations are *invalid*: $MTW = 000$ and $MTW = 001$, where M, T, and W stand for `MENUBAR`, `TOOLBAR`, and `WORDCOUNT` respectively. Therefore, no more than 6 configurations need to be run.

`SPLat` further reduces that number by dynamically analyzing the code that the test executes. For example, executing the test against the configuration $c := MTW = 100$ executes the same trace as configuration $c' := MTW = 101$. The reason is that the test only calls `createToolBar()`, which is empty in both configurations c and c' since `TOOLBAR` is false in both configurations. Although the code in `createMenuBar()` is different in c and c', the test never executes it. Therefore, having executed c, execution of c' is unnecessary. We will show in Section 3.2.3 that `SPLat` runs this test for only three configurations ($MTW = 010$, $MTW = 011$, $MTW = 100$).

3.2 Technique

Given a test for a configurable system and a complete feature model[1], `SPLat` determines all relevant configurations on which the test should be run. Each configuration run executes a unique trace of the test. `SPLat` executes the test on one configuration, observes the values of configuration variables, and uses these values to determine which configurations can be safely pruned. `SPLat` repeats this process until it explores all relevant configurations or until it reaches a specified bound on the number of configurations. As output it returns, for each test in the test suite, the configurations explored by `SPLat` for them, and the respective results (pass or fail). We first describe the feature model interface and then the core algorithm.

3.2.1 Feature Model Interface

Figure 3.3 shows the code snippet that defines the `FeatureModel` interface. The type `FeatureVar` denotes a feature variable. An object of type `Assign` encodes an assignment of boolean values to feature variables. An assignment can be *complete*, assigning values to all the features, or *partial*, assigning values to a subset of the features. A complete assignment is *valid* if it satisfies the constraints of the feature model. A partial assignment is *satisfiable*, meaning consistent (see Section 2.1.2), if it can be extended to a valid complete assignment.

```
1 class FeatureVar {...}
2 class Assign { Map<FeatureVar, Boolean> state; Stack<FeatureVar> stack; ...}
3 interface FeatureModel {
4    Set<Assign> getValid(Assign a);
5    boolean isSatisfiable(Assign a);
6    boolean isMandatory(FeatureVar v);
7 }
```

Figure 3.3: Feature Model Interface

[1] `SPLat` only worked with an incomplete or absent feature model after `SPLif`, for more detail see Section 4.2.1.

The `FeatureModel` interface provides queries for determining the validity of feature assignments, obtaining valid configurations, and checking if particular informed features are mandatory. Given an assignment α, the method `getValid()` returns the set of all complete valid assignments that assign the values of the remaining feature variables to make a valid complete assignment. Given an assignment α, the method `isSatisfiable()` checks if it is satisfiable. The method `isMandatory()` checks if a feature is mandatory according to the feature model. The existing models are written in the GUIDSL format (GUIDSL, 2013). We used the GUIDSL tool to check consistency; internally, the tool translates the input model to CNF format and uses the SAT4J SAT solver (SAT4J, 2013) to check satisfiability.

3.2.2 Main Algorithm

```
1  Map<FeatureVar, Boolean> state;
2  Stack<FeatureVar> stack;
3
4  // input, shared with instrumented code/
5  FeatureModel fm;
6
7  void SPLat(Test t) {
8   // Initialize features
9   state = new Map();
10  for (FeatureVar f: fm.getFeatureVariables())
11   state.put(f, fm.isMandatory(f));
12
13   // Instrument the code under test
14   instrumentOptionalFeatureAccesses();
15
16   do { // Repeatedly run the test
17    stack = new Stack();
18    t.runInstrumentedTest();
19    Assign pa = getPartialAssignment(state, stack);
20    print("configs covered: " + fm.getValid(pa));
21
22    while (!stack.isEmpty()) {
23     FeatureVar f = stack.top();
24     if (state.get(f)) {
25      state.put(f, false); // Restore
26      stack.pop();
27     } else {
28      state.put(f, true);
29      pa = getPartialAssignment(state, stack);
30      // it explores only satisfiable partial assignments
31      if (fm.isSatisfiable(pa))
32       break;
33     }
34    }
35   } while (!stack.isEmpty());
36 }
37
38 // called-back from test execution
39 void notifyFeatureRead(FeatureVar f) {
40  if (!stack.contains(f)) {
41   stack.push(f);
42   Assign pa = getPartialAssignment(state, stack);
43   if (!fm.isSatisfiable(pa))
44    state.put(f, true);
45  }
46 }
```

Figure 3.4: SPLat Algorithm

Figure 3.4 lists the `SPLat` algorithm. It takes as input a test `t` for a configurable system and a complete feature model `fm`. To enable exploration, the algorithm maintains a `state` that stores the values of feature variables (line 1) and a `stack` of feature variables that are read during the latest test execution (line 2).

`SPLat` performs a mostly stateless exploration of paths, that means, it does not store, restore, or compare program states as done in stateful model checking (APEL et al., 2013; D'AMORIM; LAUTERBURG; MARINOV, 2007; KäSTNER et al., 2012; KIM; KHURSHID; BATORY, 2012; RHEIN; APEL; RAIMONDI, 2011); instead, `SPLat` stores only the feature decisions made along one path and re-executes the code to explore different program paths, which corresponds to valid and dynamically reachable configurations. To that end, `SPLat` needs to be able to set the values of feature variables, to observe the accesses to feature variables during a test run, and to re-execute the test from the beginning.

The algorithm first initializes the values of feature variables (lines 8–11) using the feature model interface. Mandatory features are set to true (the only value they can have) and optional features are initially set to `false`.

A careful reader may note that initial assignment may be invalid for the given feature model. For example, initially setting feature variables to `false` would violate the constraint in our Notepad example. We describe later how `SPLat` enforces satisfiability *during* execution (in line 43). It adjusts the assignment of values to feature variables *before* test execution gets to exercise code based on an invalid configuration. Such scenario could potentially lead to a "false alarm" test failure as opposed to revealing an actual bug in the code under test. Note that the calls to `state.put()` both in the initialization block and elsewhere not only map a feature variable to a boolean value in the state maintained by `SPLat` but also set the value of the feature variable referred to by the code under test.

`SPLat` then automatically instruments (line 14) the code under test to observe feature variable reads. Conceptually, for each read of an optional feature variable (*e.g.*, reading variable `TOOLBAR` in the code `if(TOOLBAR)` from Figure 3.2), `SPLat` replaces the read with a call to the `notifyFeatureRead()` method shown in Figure 3.4 (line 39). The reads are statically instrumented so that they can be intercepted just before they happen during test execution. Mandatory feature variable reads need not be instrumented because the accessed values remain constant for all configurations.

`SPLat` next runs the test (line 18). The test execution calls the method `notifyFeatureRead` whenever it is about to read a feature variable. When that happens, `SPLat` pushes the feature variable being read on the `stack` if it is not already there, effectively recording the order of the first reads of variables. This `stack` facilitates the backtracking over the values of read feature variables.

An important step occurs during the call to `notifyFeatureRead` (line 43). The initial value assigned to the reached feature variable may make the configuration unsatisfiable. More precisely, at the beginning of the exploration, `SPLat` sets an optional feature value to

`false`. When the code is about to read the optional feature, `SPLat` checks whether the `false` value is satisfiable with the feature model, *i.e.*, whether the *partial* assignment of values to feature variables on the `stack` is satisfiable for the given feature model. If it is, `SPLat` leaves the feature as is. If not, `SPLat` changes the feature to `true`. Recall that `state.put()` sets the feature value both in the `state` map, which our algorithm uses, and in the actual boolean feature variable used in the code under test.

Note that updating a feature variable to `true` *ensures* that the new partial assignment is satisfiable. The update occurs *before* execution could have observed the old value which would make the assignment unsatisfiable. This change of value keeps the assignment satisfiable because it explores only satisfiable partial assignments (line 31), and it checks if the assignment is satisfiable in *every* variable read (line 43); thus, if a partial assignment was satisfiable considering all features on the `stack`, then it must be possible to extend that assignment with at least one value for the new feature that was not on the `stack` but is being added. If the variable associated to the new feature stores `false` at the moment execution accesses that variable, and if the partial assignment including that feature variable is *not* satisfiable, then we can change the value to `true` (line 44). Recall that optional feature variables are initialized to `false`.

After finishing one test execution for one specific configuration, `SPLat` effectively covers a set of configurations. This set can be determined by enumerating every complete assignment that (1) has the same values as the partial assignment specified by variables `state` and `stack` (line 19), and (2) is valid according to the feature model (line 20).

`SPLat` then determines the next configuration to execute by backtracking on the `stack` (lines 22–34). If the last read feature has value `true`, then `SPLat` has explored both values of that feature, and it is popped off the stack (lines 24–27). If the last read feature has value `false`, then `SPLat` has explored only the `false` value, and the feature should be set to `true` (lines 27–33). Another important step occurs now (line 31). While the backtracking over the `stack` found a partial assignment to explore, it can be the case that this assignment is not satisfiable for the feature model. In that case, `SPLat` keeps searching for the next satisfiable assignment to run. If no such assignment is found, the `stack` becomes empty, and `SPLat` terminates.

The output of the algorithm is the set of configurations covered. The side effect of the algorithm is that the test has been executed for *all* configurations that could lead to different outcomes. If any of those outcomes is a test failure, it can be printed as such.

3.2.3 Example Run

We demonstrate `SPLat` on the example from Figure 3.2. According to the feature model (Figure 3.1), `NOTEPAD` and `BASE` are the only mandatory features and are set to `true`. The other three feature variables are optional and therefore `SPLat` instruments their reads (Figure 3.2a, lines 8, 14, 23, and 28). Conceptually, the exploration starts from the configuration $MTW = 000$.

When the test begins execution, `notifyFeatureRead()` is first called when `TOOLBAR` is read. `TOOLBAR` is pushed on the stack, and because its assignment to `false` is satisfiable for the feature model, its value remains unchanged (*i.e.*, stays `false` as initialized). Had the feature model required `TOOLBAR` to be `true`, the feature's value would have been set to `true` at this point.

With `TOOLBAR` set to `false`, no other feature variables are read before the test execution finishes. In particular, `WORDCOUNT` on line 14 is not read because that line is not executed when `TOOLBAR` is `false`. Therefore, this one execution covers configurations $MTW =$ *0*, where $*$ denotes an "unknown" value. However, configurations $MTW =$ 00* are inconsistent for the given feature model, so this one execution covers two valid configurations where `TOOLBAR` is false (`MTW=100` and `MTW=101`).

`SPLat` next re-executes the test with `TOOLBAR` set to `true`, as it is satisfiable for the feature model. `WORDCOUNT` is encountered this time, but it can remain `false`, and the execution completes, covering `MTW=*10`. `SPLat` then sets `WORDCOUNT` to `true`, and the execution completes, covering `MTW=*11`. `SPLat` finally pops off `WORDCOUNT` from the stack because both its values have been explored, and pops off `TOOLBAR` for the same reason, so the exploration finishes because the stack is empty.

3.2.4 Reset Function

While a stateless exploration technique such as `SPLat` does not need to store and restore program state in the middle of execution like a stateful exploration technique does, the stateless exploration does need to be able to restart a new execution from the initial program state unaffected by the previous execution.

Restarting an execution with a new runtime (*e.g.*, spawning a new *Java Virtual Machine (JVM)* in Java) is the simplest solution, but it can be both inefficient and unsound. It is inefficient because even without restarting the runtime, the different executions may be able to share a runtime and still have identical initial program states, *e.g.*, if the test does not update any static variables in the JVM state. It can be unsound because a new runtime may not reset the program state changes made by the previous executions (*e.g.*, previous executions having sent messages to other computers or having performed I/O operations such as database updates).

We address these issues by sharing the runtime between executions and requiring the user to provide a *reset function* that can be called at the beginning of the test. One possible limitation related to this solution is the manual cost of writing a reset function, we detail this discussion in Section 3.3.6.

Our sharing of the runtime between executions means that settings that would normally be reset automatically by creating a new runtime must now be manually reset. For example, Java static initializers must now be called from the reset function because classes are loaded only once. However, we believe that the benefit of saving time by reusing the runtime outweighs the

cost of this effort, which could be alleviated by a program analysis tool.

Moreover, for the Groupon code used in our evaluation, the testing infrastructure was already using the reset function (developed independently and years before this research); between subsequent test execution, the state (of both memory and database) is reset by rolling back the database transaction from the previous test and overwriting the state changes in the `tearDown` and/or `setUp` blocks after/before each test.

3.2.5 Potential Optimization

The algorithm in Figure 3.4 is not optimized in how it interfaces with the feature model. The feature model is treated as a blackbox, read-only artifact that is oblivious to the exploration state consisting of the `state` and `stack`. Consequently, the `isSatisfiable` and `getValid` methods are executed as if the exploration state was completely new every time, even if it just incrementally differs from the previous exploration state.

For example, when running the test from Figure 3.2, `SPLat` asks the feature model if MTW = *1* is satisfiable (line 31 of the `SPLat` algorithm) after the assignment MTW = *0*. The feature model replies `true` as it can find a configuration with the feature `TOOLBAR` set to true. Then when `WORDCOUNT` is encountered while `TOOLBAR=true`, `SPLat` asks the feature model if the assignment MTW = *10 (`TOOLBAR=true` *and* `WORDCOUNT=false`) is satisfiable (line 43 of the `SPLat` algorithm). Note that the feature model is not aware of the similarity between the consecutive calls MTW = *1* and MTW = *10. In principle, it could only incrementally check the satisfiability of `WORDCOUNT=false`.

The change to the algorithm to enable this synchronization between the exploration state and the feature model can be simple: every time a feature variable is pushed on the `stack`, constrain `fm` with the feature's value, and every time a feature variable is popped off the `stack`, remove the corresponding feature assignment from the feature model. A feature model that can be updated implies that it should support incremental solving, *i.e.*, a feature model should not have to always be solved in its entirety. Our current `SPLat` tool for Java does *not* exploit incremental solving, which means that our timing results might be even better.

3.2.5.1 Feature Expression

Another way to optimize `SPLat` could be making it support feature expressions. For that, it would be necessary to create a new type representing feature expressions *FeatureExpression*, and make `SPLat` use it in its stack instead of a single feature. Now, this new type *FeatureExpression* can represent an unary expression (a single feature), or a binary expression, where each part of the expression can be another expression, allowing represent multiple expressions. The evaluation of this expression can be $true$ or $false$, but the entire expression is put in `SPLat` stack if it is satisfiable (or has no unsat core) with the previous explored features, and `SPLat` proceeds its execution normally.

3.2.6 Implementation

We implemented two versions of SPLat, one for Java, and one for C. Our collaborators also developed one version for Ruby on Rails. We selected the two languages Java and Ruby motivated by the subject programs used in our experiments (Section 3.3). The version of SPLat for C language is discussed in Chapter Section 5.

For Java, we implemented SPLat on top of the publicly available Korat solver for imperative predicates (KORAT HOME PAGE, 2013). Korat already provides code instrumentation (based on the BCEL library for Java bytecode manipulation) to monitor field accesses, and provides basic backtracking over the accessed fields. The feature variables in our Java subjects were already represented as fields. The main extension for SPLat was to integrate Korat with a SAT solver for checking satisfiability of partial assignments with respect to feature models. As mentioned earlier in Section 3.2.1, we used SAT4J (SAT4J, 2013).

For Ruby on Rails, we have an even simpler implementation because Groupon does not have feature model. So, SPLat does not prune invalid configurations, it only monitors accesses to feature variables, that are already defined in the code. We did not integrate a SAT solver, because the subject code did not have a formal feature model and thus we treated all combinations of features as valid (similar to a feature model without constraints and with all features being optional). This version of SPLat for Ruby on Rails was developed and applied by professor Darko Marinov from the University of Illinois (USA).

3.3 Evaluation

Our evaluation addresses the following research questions:

RQ1 How does SPLat's efficiency compare with alternative techniques for analyzing tests of SPLs in terms of time and number of configurations explored?

RQ2 What is the overhead of SPLat?

RQ3 Does SPLat scale to real code?

In Section 3.3.1, we compare SPLat with related techniques using 10 SPLs. In Section 3.3.5, we report on the evaluation of SPLat using an industrial configurable system implemented in Ruby on Rails.

3.3.1 Subjects

We evaluate our approach with 10 SPLs listed in Table 3.1.[2] A brief description for each is below:

[2]All subjects except 101Companies have been used in previous studies on testing/analyzing SPLs, including GPL (APEL et al., 2013; CABRAL; COHEN; ROTHERMEL, 2010), Elevator, Email, MinePump (APEL et al., 2013), JTopas by (CHANDRA et al., 2011), Notepad (KIM et al., 2010; KIM; BATORY; KHURSHID, 2011), XStream (DANIEL; GVERO; MARINOV, 2010; SCHULER; DALLMEIER; ZELLER, 2009) Prevayler by (THAKER et al., 2007; APEL; BEYER, 2011), and Sudoku (APEL; BEYER, 2011).

- **101Companies** (Human-resource management system, 2013) is a human-resource management system. Features include various forms to calculate salary and to give access to the users;
- **Email** (HALL, 2005) is an email application. Features include message encryption, automatic forwarding, and use of message signatures;
- **Elevator** (PLATH; RYAN, 2001) is an application to control an elevator. Features include prevention of the elevator from moving when it is empty and a priority service to the executive floor;
- **GPL** (LOPEZ-HERREJON; BATORY, 2001) is a product line of graph algorithms that can be applied to a graph. ;
- **JTopas** (Java tokenizer and parser tools, 2013) is a text tokenizer. Features include support for particular languages such as Java and the ability to encode additional information in a token;
- **MinePump** (KRAMER et al., 1983) simulates an application to control water pumps used in a mining operation. Features include sensors for detecting varying levels of water;
- **Notepad** (KIM et al., 2010) is a GUI application based on Java Swing that provides different combinations of end-user features, such as windows for saving/opening/printing files, menu and tool bars, etc. It was developed for a graduate-level course on software product lines;
- **Prevayler** (Library for object persistence, 2013) is a library for object persistence. Features include the ability to take snapshots of data, to compress data, and to replicate stored data;
- **Sudoku** (WEBSITE, 2013) is a traditional puzzle game. Features include a logical solver and a configuration generator;
- **XStream** (Library to serialize objects to XML and back again, 2013) is a library for serializing objects to XML and back. Features include the ability to omit selected fields and to produce concise XML for readability. XStream was converted into an SPL from an open-source Java program with the the same name.

Table 3.1 shows the number of optional features (we do not count the mandatory features because they have constant values), the number of valid configurations, and the code size for each subject SPL.

SPL	*Features*	*Confs*	*LOC*
101Companies	11	192	2,059
Elevator	5	20	1,046
Email	8	40	1,233
GPL	14	73	1,713
JTopas	5	32	2,031
MinePump	6	64	580
Notepad	23	144	2.074
Prevayler	5	32	2,844
Sudoku	6	20	853
XStream	7	128	14,480

Table 3.1: Subject SPLs.

3.3.2 Tests

We prepared three different tests for each subject SPL. The first test, referred to as `LOW`, represents an optimistic scenario where the test needs to be run only on a small number of configurations. The second test, referred to as `MED` (for MEDIUM), represents the average scenario, where the test needs to be run on some configurations. The third test, referred to as `HIGH`, represents a pessimistic scenario, where the test needs to be run on most configurations.

To prepare the LOW, MED, and HIGH tests, we modified existing tests, when available, or wrote new tests because we could not easily find tests that could be used without modification. Because some subjects were too simple, tests would finish too quickly for meaningful time measurement if test code only had one sequence of method calls. Therefore, we used loops to increase running times when necessary. Each test is a standard JUnit test that fixes all inputs except the feature variables.

We give a brief description of tests for each subject. Note that certain tests, particularly those for Email, Elevator and Mine, were run in a loop to increase the running time because they finished too quickly for meaningful time measurement. Some of the tests modify tests written, either by us or by other groups, for previously published papers. The following descriptions need not be understood in detail to understand the results.

- 101Companies's `LOW` test groups tests that exercise only one feature, each. `MEDIUM` test groups tests that exercise two features, each. `HIGH` test groups all tests;

- Elevator's `LOW` test checks the weight scale of an elevator is working correctly without moving the elevator, which means that many features are not accessed. `MEDIUM` test moves an elevator, but the test focuses on checking the maximum weight, so it does not exercise all configurations. `HIGH` test moves an elevator and exercises all configurations;

- Email's `LOW` tests public/private key look up without sending an email. Because

features are accessible only when an email is sent, this test only runs on one configuration. `MEDIUM` and `HIGH` do send emails to test certain combinations of features;

- GPL's `LOW` test only performs one algorithm, a cycle check, on the input graph, which is influenced only by search features and therefore run only on a small number of configurations. Its `MEDIUM` test also only performs one algorithm, Kruskal's, but prints the resulting graph, which accesses other feature variables and therefore runs on more configurations. Its `HIGH` test performs most of the algorithms on the input graph and therefore runs most of the configurations;

- JTopas's `LOW` test tokenizes an input without any block or line comments, so the two features that support these types of tokens are never even accessed (thus only 3 of the 5 features are accessed and only 8 configurations reachable). `MEDIUM` test uses an input without line comments, making only 16 configurations reachable. `HIGH` test tokenizes an input with both types of comments, making all configurations reachable;

- MinePump's tests exercises different sequences of method calls that result in different mine pump states and trigger varying numbers of feature variable accesses;

- Notepad's `LOW` test uses FEST user interface testing framework (FEST, 2014) to automatically launch a menuless GUI frame, simulate typing, and check that the text area shows the entered text. `MEDIUM` test performs the same check against a GUI frame with a tool bar from which different combinations of features can be exercised. `HIGH` test performs the same check against a GUI frame with both a tool bar and a menu bar, which allows two entry points to a feature and therefore exercises more configurations;

- Prevayler's `LOW` test groups tests that exercise only one feature, each. `MEDIUM` test groups tests that exercise two features, each. `HIGH` test groups all tests;

- Sudoku's `LOW` test groups tests that exercise only one feature, each. `MEDIUM` test groups tests that exercise two features, each. `HIGH` test groups all tests;

- XStream's `LOW` test constructs objects of a class without fields and serializes it to XML using an alias for the class such that only the class aliasing feature is reachable. `MEDIUM` test constructs objects of a class with fields and serializes it to XML with all but one optional feature reachable, which requires the test be run on half of the configurations. `HIGH` test is identical, except that it reaches the previously unreachable optional feature, which requires running the test on all configurations.

3.3.3 Comparison Techniques

We compared `SPLat` with different approaches for test execution. We considered two naïve approaches that run tests against *all* valid configurations: NewJVM and ReuseJVM. The ***NewJVM*** approach spawns a new JVM for each distinct test run. Each test run executes only *one* valid configuration of the SPL. It is important to note that the cost of this approach includes the cost of spawning a new JVM. The ***ReuseJVM*** approach uses the same JVM across several test runs, thus avoiding the overhead of repeatedly spawning JVMs for each different test and configuration. This approach requires the tester to explicitly provide a reset function (Section 3.2.4). Because the tester likely has to write a reset function anyway, we conjecture that this approach is a viable alternative to save runtime cost. For example, the tester may already need to restore parts of the state stored outside the JVM such as files or database.

We also compared `SPLat` with a simplified version of a previously proposed static analysis (KIM; BATORY; KHURSHID, 2011) for pruning configurations. Whereas KIM; BATORY; KHURSHID (2011) performs reachability analysis, control-flow and data-flow analyses, the simplified version, which we call ***SRA*** (Static Reachability Analysis), only performs the reachability analysis to determine which configurations are reachable from a given test. SRA builds a call graph using inter-procedural, context-insensitive, flow-insensitive, and path-insensitive points-to analysis and collects the features syntactically present in the methods of the call graph. Only the valid combinations of these *reachable* features from a test need to be run for that test.

Finally, we compared `SPLat` with an artificial technique that has zero cost to compute the set of configurations on which each test need to run. More precisely, we use a technique that gives the same results as `SPLat` but only counts the cost of executing tests for these configurations, not the cost of computing these configurations. We call this technique ***Ideal***. It gives a lower-bound for the runtime cost of `SPLat`; the overhead of `SPLat` is effectively the difference between the overall cost of `SPLat` explorations and the cost of executing tests for Ideal.

3.3.4 Results

Table 3.2 shows our results. We performed all Java experiments[3] on a machine with X86_64 architecture, Ubuntu operating system, 240696 MIPS, 8 cores, with each core having an Intel Xeon CPU E3-1270 V2 at 3.50GHz processor, and 16 GB memory. All times are listed in seconds. Our feature model implementation solves the feature model upfront to obtain all valid configurations; because this solving needs to be done for every feature model (regardless of using `SPLat` or otherwise), and because it takes a fraction of test execution time, we do not include it.

The meaning of each column in Table 3.2 is:

- **Test** identifies different categories of tests.

[3] We performed all Java experiments ten times and calculated the average.

	All Valid		SPLat				Static Reachability (SRA)		
Test	*NewJVM*	*ReuseJVM*	*Confs*	*SPLatTime*	*IdealTime*	*Overhead*	*Confs*	*Overhead*	*Time*
101Companies (192 configs)									
LOW	35.46	2.13 (6%)	32 (16%)	1.64 (77%)	0.72	0.92 (127%)	96	84.04	1.28
MED	49.37	3.9 (7%)	160 (83%)	6.84 (175%)	3.58	3.26 (91%)	192	82.54	3.99
HIGH	283.69	45.26 (15%)	176 (91%)	47.6 (105%)	41.59	6.01 (14%)	192	81.93	45.16
Elevator (20 configs)									
LOW	10.74	5.17 (48%)	2 (10%)	1.33 (25%)	0.71	0.62 (87%)	2	23.29	0.76
MED	50.97	46.65 (91%)	10 (50%)	23.62 (50%)	23.14	0.48 (2%)	20	23.74	46.17
HIGH	62.57	59.48 (95%)	20 (100%)	60.71 (102%)	59.28	1.43 (2%)	20	24.38	60.43
Email (40 configs)									
LOW	40.63	10.74 (26%)	1 (2%)	1.00 (9%)	0.87	0.13 (14%)	1	23.62	0.87
MED	57.56	48.87 (84%)	30 (75%)	36.99 (75%)	37.14	-0.15 (0%)	40	22.81	49.02
HIGH	58.02	48.93 (84%)	40 (100%)	48.96 (100%)	49.26	-0.31 (0%)	40	23.84	49.16
GPL (73 configs)									
LOW	19.21	2.23 (11%)	6 (8%)	0.79 (35%)	0.29	0.49 (168%)	6	104.97	0.30
MED	190.53	171.62 (90%)	55 (75%)	130.87 (76%)	128.52	2.35 (1%)	55	99.41	128.69
HIGH	314.20	285.89 (90%)	70 (95%)	278.77 (97%)	277.48	1.29 (0%)	73	103.52	286.28
JTopas (32 configs)									
LOW	26.59	16.83 (63%)	8 (25%)	6.29 (37%)	4.49	1.80 (40%)	32	86.87	16.44
MED	29.04	18.55 (63%)	16 (50%)	13.16 (70%)	9.71	3.46 (35%)	32	86.87	18.70
HIGH	28.92	18.93 (65%)	32 (100%)	25.31 (133%)	18.43	6.88 (37%)	32	86.87	18.48
MinePump (64 configs)									
LOW	23.71	7.53 (31%)	9 (14%)	3.65 (48%)	1.90	1.75 (91%)	64	22.69	7.49
MED	59.72	14.78 (24%)	24 (37%)	10.43 (70%)	6.26	4.17 (66%)	64	22.38	15.35
HIGH	13.72	5.75 (41%)	48 (75%)	37.80 (657%)	4.81	32.99 (685%)	64	22.18	5.77
Notepad (144 configs)									
LOW	398.22	135.60 (34%)	2 (1%)	3.06 (2%)	2.45	0.61 (24%)	144	80.40	135.47
MED	418.23	156.27 (37%)	96 (66%)	104.95 (67%)	104.91	0.04 (0%)	144	80.62	156.35
HIGH	419.99	153.39 (36%)	144 (100%)	153.11 (99%)	152.16	0.94 (0%)	144	81.29	151.94
Prevayler (32 configs)									
LOW	65.34	40.23 (61%)	12 (37%)	22.49 (55%)	22.8	-0.31 (-1%)	32	205.54	45.39
MED	121.38	96.5 (79%)	24 (75%)	102.49 (106%)	105.86	-3.37 (-3%)	32	214.67	111.37
HIGH	149.08	120.7 (80%)	32 (100%)	127.17 (105%)	131.37	-4.2 (-3%)	32	290.66	135.61
Sudoku (20 configs)									
LOW	51.11	48.1 (94%)	4 (20%)	42.72 (88%)	24.12	18.6 (77%)	10	31.87	24.28
MED	118.14	105.67 (89%)	10 (50%)	58.31 (55%)	54.16	4.15 (7%)	10	31.75	53.67
HIGH	489.6	334.82 (68%)	20 (100%)	316.47 (94%)	332.36	-15.89 (-4%)	20	31.74	338.48
Xstream (128 configs)									
LOW	111.26	30.04 (27%)	2 (1%)	1.57 (5%)	1.08	0.49 (45%)	2	106.50	1.06
MED	105.10	9.04 (8%)	64 (50%)	5.77 (63%)	5.26	0.51 (9%)	64	109.22	5.14
HIGH	101.66	8.68 (8%)	128 (100%)	9.16 (105%)	8.59	0.57 (6%)	128	105.68	8.74

Table 3.2: Experimental Results for Various Techniques

- **All Valid** identifies the techniques that run the test against all valid configurations, namely **NewJVM** and **ReuseJVM**. **ReuseJVM** shows time absolutely and as a percentage of **NewJVM** duration.
- Columns under **SPLat** details information for SPLat:
  - **Confs** shows the number of configurations that SPLat runs for a particular test. The value in parentheses shows the percentage of configurations that SPLat explores compared to the total;
  - **SPLatTime** shows the runtime cost of SPLat. We use the same JVM for running tests. The value in parentheses shows the percentage of the time of **ReuseJVM** (not **NewJVM**);
  - **IdealTime** shows the time in seconds for running SPLat without considering the cost to determine which configurations to run for the tests; therefore, this number excludes instrumentation, monitoring, and constraint solving;

- **Overhead** shows the overhead of `SPLat`, calculated by subtracting **IdealTime** from **SPLatTime**, and dividing it by **IdealTime**.

- Columns under **Static Reachability (SRA)** show results for our static analysis:
  - **Confs** shows the number of configurations reachable with such analysis;
  - **S.A.Overhead** shows the time taken to perform the static reachability analysis;
  - **S.A.Time** shows the time taken to run the configurations determined by this analysis.

Efficiency.

The **ReuseJVM** column shows that reusing JVM saves a considerable amount of time compared to spawning a new JVM for each test run. For example, for half of the tests, reusing JVM saves over 50% of the time, because running these tests does not take much longer than starting up the JVM. For tests that take considerably longer than starting up the JVM, such saving is not possible.

`SPLat` further reduces the execution time over **ReuseJVM** by determining the reachable configurations. For example, for the `LOW` test for `Notepad`, reusing the JVM takes 34% of the time to run without reusing the JVM, and with `SPLat`, it takes just 2% of the already reduced time. In fact, the table shows that in *most* cases, as long as `SPLat` can reduce the number of configurations to test (*i.e.*, **Confs** is lower than the total number of configurations), it runs faster than running each configuration (*i.e.*, less than 100% of **ReuseJVM**).

Comparison with Static Reachability Analysis.

The Static Reachability Analysis yields less precise results compared to `SPLat`: the number of configurations in the column **Confs**, in most cases, is larger than the corresponding number of configurations in the corresponding column for `SPLat`. In some cases there is a tie, but SRA is never better than `SPLat`. In fact, for `JTopas`, `Notepad` and `MinePump`, the SRA reports all features as being accessed from the call graph, and therefore reports that all valid configurations have to be tested. For example, for `JTopas`, this is due to its tests invoking the `main` method of the SPL, from which all feature variable accesses may be reached using different input values, which the analysis is insensitive to.

For `Notepad`, this is due to the the use of the `FEST` automated GUI testing framework, which relies heavily on reflection. Because the method being invoked through reflection cannot necessarily be determined statically, the analysis yields a conservative result. For `MinePump`, each test happens to exercise a sequence of methods that together reach all feature variable accesses.

Note that the SRA approach first statically determines the configurations to run (which takes the time in column **SRA Overhead**) and afterwards dynamically runs them one by one

(which takes the time in column **SRA Time**). Comparing just the static analysis time (**SRA Overhead**) with the SPLat overhead (**SPLat Overhead**) shows that SRA has a considerably larger overhead, in some cases two order of magnitudes longer.

Although the static analysis overhead can be offset by using the reachable configurations it determines against tests that have the same code base but have different inputs, in general, it would require a very large number of such tests for the approach to have a smaller overhead than SPLat. Moreover, comparing just the time to execute the configurations computed by SRA (column **SRA Time** with the time to execute the configurations computed by SPLat (column **IdealTime**) shows that SRA again takes longer, typically proportional to the higher number of configurations that it conservatively computes.

RQ1. Based on the comparison with **NewJVM**, **ReuseJVM**, and **SRA**, we conclude the following:

> In most cases, SPLat is more efficient than the techniques that run all valid configurations for tests or prune reachable configurations using static analysis. Moreover, compared with static analysis, SPLat gives results faster and more precise in most cases.

Overhead.

Table 3.2 also shows the overhead that SPLat has over the Ideal technique (column **SPLat Overhead**). The overhead is generally small, except for the LOW tests and tests for several subjects (e.g. JTopas and Mine). The overhead is high for the LOW tests because these tests finish quickly (under 7 seconds, often under 1 second), meaning that instrumentation, monitoring and feature model interaction take a larger fraction of time than they would for a longer executing test. The overhead is high for JTopas because the feature variables are accessed many times because they are accessed within the tokenizing loop. (A rewrite approach that would unroll the loop could reduce this overhead.) The overhead is high for MinePump because feature accesses and their instrumentation take relatively longer to execute for this particular test as the subject is very small.

SPLat, due to its cost in monitoring feature variables, should not execute a test faster than knowing the reachable configurations upfront and running the test only on those configurations. Thus, the occasional small negative overheads for Email and Prevayler are due to the experimental noise and/or the occasionally observed effect where an instrumented program runs faster than the non-instrumented program. It is important to note that the efficiency and overhead are orthogonal. As long as the reduction in time due to the reduction in configurations is larger than the overhead, SPLat saves the overall time. To illustrate, the GPL's LOW test incurs over 168% overhead, but the reduction in configurations outweighs the overhead, and SPLat takes only 35% of running all valid configurations with the same JVM.

RQ2. Based on the discussion about overhead, we conclude the following:

`SPLat` can have a large relative overhead for short-running tests, but the overhead is small for long-running tests.

3.3.5 Case Study: Groupon

Groupon is a company that "features a daily deal on the best stuff to do, see, eat, and buy in 48 countries" (`http://www.groupon.com/about`). *Groupon PWA* is the name of the codebase that powers the main groupon.com website. It has been developed for over 4.5 years with contributions from over 250 engineers. The server side is written in Ruby on Rails and has over 171K lines of Ruby code.

Groupon PWA code is highly configurable with over 170 (boolean) feature variables. In theory, there are over 2^{170} different configurations for the code. In practice, only a small number of these configurations are ever used in production, and there is one default configuration for the values of all feature variables.

Groupon PWA has an extensive regression testing infrastructure with several frameworks including Rspec, Cucumber, Selenium, and Jasmine. The test code itself has over 231K lines of Ruby code and additional code in other languages. (It is not uncommon for the test code to be larger than the code under test (TILLMANN; SCHULTE, 2005).)

Groupon PWA has over 19K Rspec (unit and integration) tests. A vast majority of these tests run the code only for the default configuration. A few tests run the code for a non-default configuration, typically changing the value for only one feature variable from the default value. Running all the Rspec tests on a cluster of 4 computers with 24 cores each takes under 10 minutes.

3.3.5.1 SPLat Application

Our collaborators implemented `SPLat` for Ruby on Rails to apply it to Groupon PWA. We did not have to implement the reset function because it was already implemented by Groupon testers to make test execution feasible (due to the high cost of re-starting the system). Moreover, no explicit feature model was present, so feature model constraints did not need to be solved.

3.3.5.2 Results

We set out to evaluate how many configurations each test could cover if we allow varying the values of all feature variables encountered during the test run. We expected that the number of configurations could get extremely high for some tests to be able to enumerate all the configurations. Therefore, we set the limit on the number of configurations to not more than 17 (10% from the total number of features), so that the experiments can finish in a reasonable time[4]. This limit was reached by 2,695 tests. For the remaining 17,008 tests, Figure 3.5 shows

[4]Since this study of Groupon was performed by professor Darko Marinov, and he did not record the time the experiments took to run, it is not possible to show the exact time, he only registered that was a reasonable time.

the breakdown of how many tests reached a given number of configurations. We can see that the most common cases are the number of configurations being powers of two, effectively indicating that many features are encountered independently rather than nested (as in Figure 3.2 where the read of `WORDCOUNT` is nested within the block controlled by the read of `TOOLBAR`).

Confs.	*Tests*
1	11,711
2	1,757
3	332
4	882
5	413
6	113
7	19
8	902
9	207

Confs.	*Tests*
10	120
11	29
12	126
13	6
14	32
15	10
16	349
17	2,695
-	-

Figure 3.5: The distribution of the number of reachable configurations per tests, Figure 3.6 details the distribution of the number of features per tests.

We also evaluated the number of features encountered. It ranges from 1 up to 43. We found 43 is a high number in the absolute sense (indicating that a test may potentially cover 2^{43} different configurations), 43 is also a relatively low number in the relative sense compared to the total of over 170 features. Figure 3.6 shows the breakdown of how many tests reached a given number of feature variables. Note that the numbers of configurations and feature variables may seem inconsistent at a glance, *e.g.*, the number of tests that have 1 configuration is larger than the number of tests than have 0 feature variables. The reason is that some tests force certain values for feature variables such that setting the configuration gets overwritten by the forced value.

Vars	*Tests*
0	11,711
1	1,757
2	1,148
3	1,383
4	705
5	389
6	466
7	323
8	425
9	266
10	140
11	86
12	80
13	34

Vars	*Tests*
14	28
15	54
16	62
17	1
19	14
20	260
21	109
22	45
23	19
24	22
25	9
26	2
27	14
28	17

Vars	*Tests*
29	6
30	8
31	24
32	6
33	14
34	31
35	11
36	15
37	8
38	2
39	2
40	3
41	2
42	2

Figure 3.6: The distribution of the number of features accessed per tests.

In sum, these results show that the existing tests for Groupon PWA can already achieve a high coverage of configurations, but running all the configurations for all the tests can be

prohibitively expensive. We leave it as a future work to explore a good strategy to sample from these configurations (COHEN; DWYER; SHI, 2006, 2007; MCGREGOR, 2001).

RQ3. Moreover, based on the fact that it was possible run `SPLat` on the codebase as large as Groupon PWA in a reasonable time, we conclude the following:

> Considering the results, `SPLat` scales to the setup of this experiment, providing evidence that `SPLat` is able to scale to a large industrial code. The implementation effort for `SPLat` is relatively low and the number of configurations covered by many real tests is relatively low.

3.3.6 Threats to Validity

The main threat is to external validity: we cannot generalize our timing results to all SPLs and configurable systems because our case studies may not be representative of all programs, and our tests may be covering an unusual number of configurations. To reduce this threat, we used multiple Java SPLs and one real, large industrial codebase. For SPLs, we designed tests that cover a spectrum of cases from `LOW` to `MED(IUM)` to `HIGH` number of configurations, even with this threat, these spectrum of cases and the modified tests may be insufficient or inappropriate. For the Groupon codebase, we find that most real tests indeed cover a small number of configurations.

Another threat is that `SPLat` does not support concurrent programs, because they lead to many states in the program and, consequently, to non-deterministic test results. One way to mitigate this threat would be, in addition to have a stack to monitor the global state of the features, to add extra stacks in order to monitor each concurrent part of the program.

Our study has the usual internal and construct threats to validity. `SPLat` is a helpful technique that can be used in practice to improve testing of SPLs. An important threat to this conclusion is that our results do not take into account the manual cost of writing a reset function. NewJVM, ReuseJVM, and `SPLat` all require the state outside of the JVM to be explicitly reset, but NewJVM automatically resets JVM-specific state by spawning a new JVM for each test run. We expect that the cost of writing the reset function can be amortized over multiple tests and test runs.

4

SPLif

Feature models capture dependencies among features and can (1) reduce the space of programs to test and (2) enable accurate categorization of failing tests as failures of programs or the tests themselves, not as failures due to inconsistent combinations of features. In practice, unfortunately, feature models are not always available.

We introduce `SPLif`, the first approach for testing software product lines that does not require a priori availability of feature models. Our insight is to use a profile of passing and failing test runs to quickly identify test failures that are indicative of a problem (in test or code) as opposed to a manifestation of execution against an inconsistent combination of features. Experimental results demonstrate the effectiveness of our approach.

4.1 Illustrative Example

The main problem for testing SPLs with incomplete feature models (in the limit empty models) is that if a test fails it is difficult to determine with absolute certainty if there is a bug in the code, a bug in the test, or the bug manifests because of an inconsistent combination of features. We propose `SPLif` to address this problem.

4.1.1 Notepad

To illustrate, we use `Notepad`, a simple visual text editor that has been previously used in related research (KIM et al., 2010; KIM; BATORY; KHURSHID, 2011; KIM et al., 2013). `Notepad` has 2,074 lines of code, 17 features, and 62 tests. While `Notepad` has a complete feature model formally specified, for the sake of illustration, we assume here an empty model.

4.1.2 A Notepad test (failure)

Figure 4.1 shows a test for `Notepad`. This test is implemented using the GUI testing framework FEST (FEST, 2014) and uses buttons available from the toolbars to cut and paste a string written to the text area. To find these buttons, this test uses the auxiliary function

getButtonByToolTip (line 15), which iterates over all buttons reachable from the parent window of the text area until finding one that matches the given tooltip text, passed as parameter. This test fails when the feature *UNDOREDOTOOLBAR* is enabled. That feature adds to the `Notepad` window a new toolbar containing the buttons "undo" and "redo"; they become reachable in the search for buttons. The failure manifests because this feature initializes the *toolTipText* field of these buttons with **null**. Consequently, execution raises *NullPointerException* at line 20 when trying to derreference this field from the "undo" button.

```
1  public void testEditToolBar() {
2   JTextComponentFixture textArea = window.textBox();
3   textArea.enterText("Hi");
4   JButtonFixture cutButton = getButtonByTooltip("Cut", window);
5   cutButton.requireVisible();
6   cutButton.requireEnabled();
7   textArea.selectAll();
8   cutButton.click();
9   assertThat(textArea.text()).contains("");
10  JButtonFixture pasteButton = getButtonByTooltip("Paste", window);
11  pasteButton.click();
12  pasteButton.click();
13  assertThat(textArea.text().contains("HiHi");  } }
14
15 private JButtonFixture getButtonByTooltip(String toolTipText, ContainerFixture frame){
16  GenericTypeMatcher<JButton> buttonMatcher =
17    new GenericTypeMatcher<JButton>(JButton.class){
18      @Override
19      protected boolean isMatching(JButton button){
20       return button.getToolTipText().equals(msg);
21      }
22    };
23  return frame.button(buttonMatcher); }
```

Figure 4.1: Test of `Notepad` that checks the cut and paste functionalities. This test fills the text area with a string, selects the text area, and presses the "cut" button. It then asserts that the text area is indeed empty, presses the "paste" button twice, and checks for the presence of the repeated string.

4.1.3 SPLat

`SPLif` builds on `SPLat` (KIM et al., 2013), a technique that we previously developed, to execute each test several times, once for each configuration that a test encounters during execution. *Failing tests* are tests whose executions failed on at least one configuration. We call such executions *failing runs*, and call configurations on which tests fail *failing configurations*. With `Notepad` and an empty model, `SPLat` explores all given 62 tests, a total of 5,094 runs (excluding time-outs) on distinct configurations. Out of these 5,094 runs, 300 were failing runs that were attributed to 3 tests on different configurations. It is daunting to inspect all 300 failures in order to find real problems. `SPLif` reduces the number of inspections to 10.

4.1.4 SPLif in a nutshell

`SPLif` takes as input an SPL with its test suite and incomplete feature model. It explores each test with various configurations, and produces a ranked list of failing tests and

configurations for the user to inspect. Inspired by Tarantula (JONES; HARROLD; STASKO, 2002), `SPLif` *ranks tests* based on the *ratio* of failing and passing configurations. `SPLif` also *ranks configurations* using various strategies to prioritize the order in which they should be inspected. An inspection can result in a repair of the test, a repair of the feature model, and/or a repair of the code under test. If a test fails on a consistent configuration, a real problem is revealed in the target source code or in the test itself.

If a test fails on an inconsistent configuration, the incomplete feature model can be updated to incorporate the violated constraint, tests that pass for all the unknown or known consistent configurations do not appear in the ranking that `SPLif` reports. The user proceeds inspecting failing tests and configurations until a budget limit is reached; the goal is to quickly find fault-revealing consistent configurations that cover (all) failing tests.

4.1.5 SPLif on Notepad Tests

With an empty feature model, `SPLat` explores all given 62 tests of `Notepad` and a total of 5,094 runs. Out of these 5,094 runs, 300 were failing and were attributed to 3 tests (Figure 4.6) on 265 distinct configurations.

Most failing runs can be attributed to illegal configurations. We want to find failing configurations for *legal* configurations. If a user randomly inspected the 133 failed runs for *testEditToolBar*, (s)he would examine 37 configurations (on average) before arriving at the first consistent configuration. `SPLif` does better than this: by following its ranking, the first consistent configuration is found in the first 10 inspections. By prioritizing tests and configurations to inspect, `SPLif` makes users more productive.

Proceeding with this process until all failing tests have been inspected, a random ranking would require a total of 90 inspections, and a ranking generated by `SPLif` would require no additional inspection, because the same consistent configurations that `SPLif` reports for the first test inspected is reachable by the other 2 failing tests as well. So, `SPLif` detects that all 3 tests expose a bug in the test code or in the source code by inspecting 10 configurations. This result represents a reduction of 88.9% compared to random inspection, and a reduction of 96.7% compared to inspect all 300 failing configurations (Figure 4.9), in the worst case.

4.2 Technique

`SPLif` takes as input a test suite and an incomplete feature model for an SPL and reports as output a ranked list of failing tests sorted by their likelihood of containing consistent failing configurations. For such test from the input test suite, `SPLif` reports a ranked list of configurations for inspection by their likelihood of being consistent. These rankings expose real problems in code or tests more quickly.

`SPLif` has three fully automated parts: Test Exploration (Section 4.2.1), Test Ranking

(Section 4.2.2), and Configuration Ranking (Section 4.2.3). The overall approach consists of the following steps:

1. Use `SPLif` to run a given test suite with an incomplete or empty feature model (Section 4.2.1).
2. Use `SPLif` to rank tests (Section 4.2.2).
3. Pick the highest ranked test to inspect.
4. Use `SPLif` to rank failing configurations for that test (Section 4.2.3).
5. Pick the highest ranked configuration to inspect.
6. If the configuration is inconsistent, the user can provide that information about the configuration to `SPLif` to make the feature model more complete.
7. If the configuration is consistent, the user should repair the test or the code under test.
8. Repeat steps 1-7 until running out of time budget or finishing the inspection of all failing tests.

Note that we could have chosen to inspect a configuration that fails for multiple tests and then sort tests to inspect. We prioritized tests and then configurations because we found it more intuitive to focus on one test at a time and to reason about multiple configurations for that one test.

4.2.1 Test Exploration

`SPLif` uses `SPLat` (KIM et al., 2013) to obtain, for each test, all (not known to be inconsistent) configurations that have a unique trace during the test execution. We described `SPLat` in detail in previous chapter, so we summarize it only briefly here, and then describe the changes we made for `SPLif`.

Given an SPL code base, a test and a *complete* feature model (we discuss incomplete models in the next paragraph), `SPLat` executes the test on one configuration, observes the values of feature variables that have been accessed during the execution, and uses these values to determine what other configurations should be considered in subsequent test executions. For example, if a test execution accessed only one feature variable, f, with value $false$, then `SPLat` re-executes the test with f set to $true$. If that second execution accesses no other feature variables, the search stops. Otherwise, it continues to explore the combinations of values of the other accessed variables. `SPLat` repeats this process until it explores all dynamically reachable configurations or until it reaches a configurations.

In previous chapter, `SPLat` assumes that, in addition to the test, a *complete* feature model is provided as input. This allows `SPLat` to substantially reduce the space of possible configurations to explore. In contrast, `SPLif` assumes an *incomplete* feature model, so we had to change `SPLat` to treat every unknown configuration as valid (✓), if the model is complete, or as consistent, if the model is incomplete. When `SPLat` runs now, it prunes test execution paths that

correspond to definitely inconsistent configurations and explores all other paths. Additionally, we changed `SPLat` to accept a time limit for each test execution, because executing inconsistent configurations can take a long time or even lead to infinite loops.

4.2.2 Test Ranking

`SPLif` computes a ranking of tests for all executions with unknown configurations. For inconsistent configurations, `SPLif`/`SPLat` does not even execute a test. For consistent configurations, the situation is quickly resolved: if the test passes, the execution is ignored; if the test fails, the execution immediately shows a real problem in the test or the code.

For each test t, we define the following terms:

- $P_t \stackrel{\text{def.}}{=}$ number of passing unknown configurations of t,
- $F_t \stackrel{\text{def.}}{=}$ number of failing unknown configurations of t,
- $FC_t \stackrel{\text{def.}}{=}$ number of consistent configurations for which t fails,
- $S_t \stackrel{\text{def.}}{=}$ *suspiciousness rating* or fraction of failed unknown configurations. (*i.e.*, $S_t = \frac{F_t}{(F_t+P_t)}$).

Ranking is obtained by lexicographically sorting the triples $\langle FC_t,\ S_t,\ F_t \rangle$ associated with each test t. Spectrum-based fault localization algorithms (*e.g.*, Tarantula (JONES; HARROLD; STASKO, 2002) and Barinel (ABREU; ZOETEWEIJ; GEMUND, 2009)) use a similar criterion to classify suspicious statements.

Tests that appear higher in this ranking are considered more likely to contain a consistent configuration $FC_t > 0$. Note that if a test fails for a consistent configuration, we know it has a bug (in code or test) and that test needs inspection irrespective of the number of consistent configurations it succeeds or fails.

The metrics S_t and F_t differentiate tests in the middle of the ranking. In case of ties on metric FC, test t_1 will be ranked higher than t_2 if $S_{t1} > S_{t2}$. The rationale for using S_t is that there is a higher chance of finding a failing consistent configuration when `SPLat` reaches a relatively high number of failing configurations. Finally, if the first two metrics do not differentiate t_1 and t_2, `SPLif` uses the third metric as a tie-breaker ($F_{t1} > F_{t2}$).

`SPLif` can rank tests once at the beginning of its execution (option `STATIC`) or rerank tests after every test inspection (option `DYNAMIC`). Intuitively, reranking could help `SPLif` to better categorize the tests remaining for inspection: after the user labels the configurations from one test, the feature model becomes more complete. So if inspecting one test determines that some configuration is inconsistent, then the same configuration can be ignored for all other tests. Likewise, if inspecting one test determines that some configuration is consistent, and another test fails for that configuration, then that other test immediately shows a real problem and goes to the top of the ranking.

4.2.3 Configuration Ranking

The previous section explains how SPLif ranks tests. Now, we explain how SPLif ranks configurations associated with a given test. SPLif has two options to rank configurations, namely MEMORY and WEIGHTED. If no option is selected, SPLif randomly orders configurations for inspection. If MEMORY is set, SPLif uses previously labeled configurations (configurations classified by the user as consistent or inconsistent) to update the incomplete feature model that SPLif maintains, such that configurations whose labels can be inferred from the model are not inspected again (lines 27–32 of Figure 4.2), they are removed from the configuration ranking. Note that MEMORY does not define an explicit order of configurations. If WEIGHTED is set, SPLif sorts the configurations according to three different criteria.

For each configuration c, we define the following terms:

- $?_c \stackrel{\text{def.}}{=}$number of undefined feature variables in c,
- $F_c \stackrel{\text{def.}}{=}$number of failing tests that execute c, and
- $SC_c \stackrel{\text{def.}}{=}$boolean that indicates if c is *similar* to some previously seen consistent configuration.

If the option WEIGHTED is set, the ranking of configurations is obtained by lexicographically sorting the triples $\langle ?_c,\ F_c,\ SC_c \rangle$ associated with each configuration c. Configurations that appear higher in the ranking are considered more helpful to improve overall performance of SPLif.

The first element of the triple helps to rank higher those configurations that could be obtained with instantiations of undefined variables, and the second element helps to rank higher those configurations that occur in yet-to-be-inspected tests. The rationale for these metrics is to optimize future labelings (C or I) of configurations. For example, if a configuration is inconsistent, all complete extensions of it must be inconsistent; hence it is beneficial to label such configurations quickly as they label more complete configurations.

The third element ranks higher those configurations that look similar to consistent configurations. Similarity is determined by checking if f_c is satisfiable with $\mathscr{M}_C$, which suggests that the configuration is in accordance with the current feature model learned from the previously labeled consistent configurations. Note, however, that similarity to consistent configuration does not imply consistency (see Example from Section 2.1). In case of ties in the ranking SPLif uses random ordering.

4.2.4 Algorithm

Figure 4.2 shows the pseudo-code of SPLif. It takes as input a test suite T for a software product line and its incomplete feature model.

We represent the feature model as two sets of configurations, one encoding consistent configurations ($\mathscr{M}_C$) and one encoding inconsistent configurations ($\mathscr{M}_I$). We can view each set

```
1  /* summary of SPLat execution for a test */
2  class TestInfo { Test, Set<Conf> pass, fail; }
3
4  /* models and test suite */
5  INPUTS: M_C, M_I, T
6
7  SPLif()
8
9   /* collect test profiles */
10  Set<TestInfo> Π = ∅
11  foreach t_i in T do
12    Π = Π ∪ {SPLat(M_I, t_i) /*test info*/}
13  R = list(Π)
14
15  /* rank tests by their execution profiles */
16  if (STATIC) R = rankTests(R)
17
18  attest:
19  while R ≠ ∅ do
20
21   /* dynamically (re)rank tests */
22   if (DYNAMIC) R = rankTests(R)
23   t = head(R); R = tail(R)
24
25   /* inspect test t if c has been previously
26        inspected and labeled as consistent */
27   if ({c : t.fail | M_C ∩ {c}} ≠ ∅)
28     break  /* inspect test! */
29
30   /* f_k is the logical encoding of configuration k */
31   Set<Conf> Δ = {k : t.fail | isSAT(f_k ∧ ¬M_I)}
32   if (Δ = ∅) continue /* ignore test! */
33
34   /* ranking confs. with unknown labels */
35   while Δ ≠ ∅
36     c = head(rankConfs(Δ))
37     Δ = Δ - {c}
38     switch ulabel(c) /* user labels c */
39       case V:
40         if (MEMORY)
41           M_C = M_C ∨ f_c
42         break attest  /* inspect test! */
43       case I:
44         if (MEMORY)
45           M_I = M_I ∨ f_c
46           /* update set of unknown configurations */
47           Δ = {k : Δ | isSAT(f_k ∧ ¬M_I)}
48         break
49     update(c, T);
50
51 /* ranking tests*/
52 List<TestInfo> rankTests(List<TestInfo> R)
53  return sortLexicographically(R, λt :Test.⟨FC_t, S_t, F_t⟩)
54
55 /* ranking configurations*/
56 List<Conf> rankConfs(Set<Conf> Δ)
57  if (WEIGHTED)
58   return sortLexicographically(Δ, λc :Conf.⟨?_c, F_c, SC_c⟩)
59  else return randomOrder(Δ)
```

Figure 4.2: The `SPLif` Algorithm.

as a formula, a disjunction of the (conjunctive) formulas that represent each configuration in the set. A well-formed feature model should have no overlap between consistent and inconsistent sets of configurations, *i.e.*, $\mathscr{M}_C \wedge \mathscr{M}_I$ should be unsatisfiable. A feature model is empty if both $\mathscr{M}_C$ and $\mathscr{M}_I$ are empty, and a feature model is complete if $\mathscr{M}_C \vee \mathscr{M}_I$ is equivalent to *true*.

At line 12 the algorithm calls `SPLat`, which returns results for each test in the test suite.

For example, it outputs the results (pass or fail) for each configuration that `SPLat` explores on a given test. The algorithm proceeds by iteratively choosing the top ranked test and then focusing on the failing configurations for each selected test.

There are two strategies for resorting the ranking of tests. The basic mode (option `STATIC`) sorts all the tests at the beginning of execution whereas the adaptive mode (option `DYNAMIC`) resorts the remaining tests after each test is inspected.

If any failing configuration is already in the consistent set, then the test shows a real problem that should be repaired (lines 27–28). If that does not hold but all configurations are in the inconsistent set, then the test should be ignored (line 32). Otherwise, the algorithm iterates through the set of still unknown configurations (lines 19–49). The algorithm sorts these configurations using one of the strategies discussed in Section 4.2.3.

Finally, the algorithm picks the highest ranked configuration and asks the user to label it. If the configuration is consistent, this scenario reveals a real problem, and the user should stop the inspection of additional configurations to address the problem. If the configuration is inconsistent, it is added to the set of inconsistent configurations, and the inspection for this test proceeds to the next configuration. Note that the set of failing unknown configurations (Δ) is updated accordingly. The inspection finishes after the user finds a consistent configuration (as that is a clear indication of a real problem in code or test) or inspects all the inconsistent configurations for each failing test.

The call to `update` (line 49) updates the counts of passing and failing configurations (*i.e.*, P_t and F_t for each test t from T) after every configuration labeling. In `DYNAMIC` mode, this update can potentially modify the relative ranking of each test in $\mathscr{R}$.

4.3 Evaluation

The primary goal of `SPLif` is to provide better support for testing SPLs. Based on that goal we pose the following research questions:

RQ1 How does `SPLif` rank faulty tests for inspection?

RQ2 How does `SPLif` rank configurations (of selected tests) for inspection?

4.3.1 Subjects

We *initially* evaluated `SPLif` using five small subjects (size range: 1.7–3.6KLOC) that were used to test SPLs previously. Figure 4.3 tabulates, for each subject, the source of the subject, the number of feature variables, the number of valid (complete) configurations, and the code size.

Subject	*Source*	*#Features*	*#Valid Confs.*	*LOC*
Companies	Human-resource management system (2013)	10	192	2,059
DesktopSearcher	LENGAUER et al. (2013)	16	462	3,779
GPL	LOPEZ-HERREJON; BATORY (2001)	13	73	1,713
Notepad	KIM et al. (2010)	17	256	2,074
ZipMe	APEL; BEYER (2011)	13	24	3,650

Figure 4.3: Target SPLs used in SPLif evaluation.

4.3.2 Setup

4.3.2.1 Tests analyzed

Since we do not have good feature coverage with the existing tests, and existing tools[1] generates tests with a poor feature coverage, we asked 5 students, with experience in testing, to create tests. We assigned two subjects per student and two students per subject, and provided instructions on how to create tests. In summary, we instructed students to aim for creating a test suite that achieves maximum feature coverage. Detailed instructions that students received for creating tests can be found elsewhere (Instructions to students on generating new tests for SPLs., 2014). We provided support to students on how to use the FEST framework citepfest to create GUI tests for DesktopSearcher and Notepad.

Figure 4.4 shows, for each SPL, the number of tests and the number of configurations that those tests dynamically reach. Column "*All*" shows the total number of reachable configurations and column "*>1 Failing*" shows the number of reachable configurations that cause at least one test to fail. Values in parentheses denote the number of consistent configurations in the corresponding set. Figure 4.5 shows the distributions of consistent configurations (column "*CCs*") vs number of failing tests (column "*FTs*").

		Reachable Confs.			
Subject	*# Tests*	*All*		*>=1 Failing*	
Companies	19	152	(64)	72	(4)
GPL	25	1,268	(137)	497	(49)
Notepad	62	3,273	(66)	265	(6)
DesktopSearcher	44	254	(160)	125	(83)
ZipMe	62	2,431	(153)	261	(12)

Figure 4.4: Tests and Reachable Configurations. Values in parentheses show the subset of consistent configurations.

For example, 14 configurations of GPL fail in only one test (but not necessarily the same test). Figure 4.6 shows, for each SPL and test, the number of configurations that make the test run fail (column "*F*"), the number of configurations that the test passes (column "*P*"), and the number of consistent configurations that produce failure (column "*FC*"; $FC \leq F$). Each table

[1] For example, Randoop (https://code.google.com/p/randoop/) and Evosuite (http://www.evosuite.org/).

contains two sections: the first section includes entries where *FC=0* and the second where *FC≠0*. We omit test entries that only pass, *i.e.*, entries where $F = 0$. It is worth noting that in this experiment we initialize `SPLif` with an empty feature model; the purpose of existing models is to validate configurations.

Companies

CCs	*FTs*
4	1

GPL

CCs	*FTs*
14	1
35	2

Notepad

CCs	*FTs*
5	1
1	3

DesktopSearcher

CCs	*FTs*
66	1
9	2
7	3
1	4

ZipMe

CCs	*FTs*
12	1

Figure 4.5: Distribution of number of consistent configurations (*CCs*) per number of failing tests (*FTs*).

Companies

t_i	F_i	P_i	FC_i
13	16	16	0
14	24	8	0
15	48	16	0
16	48	16	0
17	1	1	1
18	7	5	1
19	12	4	2

GPL

t_i	F_i	P_i	FC_i
18	4	4	0
19	4	4	0
20	33	78	0
21	192	64	0
22	7	1	1
23	26	26	2
24	248	8	35
25	256	256	46

Notepad

t_i	F_i	P_i	FC_i
29	67	62	1
30	100	138	4
31	133	94	3

DesktopSearcher

t_i	F_i	P_i	FC_i
10	1	4	0
11	1	8	0
12	1	8	0
13	1	2	0
14	1	8	0
15	1	8	0
16	1	8	0
17	1	8	0
18	1	32	0
19	1	8	0
20	1	32	0
21	1	4	0
22	1	8	0
23	1	32	0
24	1	0	0
25	1	0	0
26	2	8	0
27	3	8	0
28	3	8	0
29	3	2	0
30	3	8	0
31	1	25	1
32	3	1	3
33	3	1	3
34	3	1	3
35	4	0	3
36	5	0	4
37	7	1	7
38	7	1	7
39	14	0	8
40	14	1	13
41	15	2	14
42	17	1	16
43	28	3	27

ZipMe

t_i	F_i	P_i	FC_i
30	1	1	0
31	1	1	0
32	1	3	0
33	1	3	0
34	1	8	0
35	1	2	0
36	1	3	0
37	1	3	0
38	1	3	0
39	1	3	0
40	1	3	0
41	1	2	0
42	1	1	0
43	1	3	0
44	1	3	0
45	2	12	0
46	2	8	0
47	2	6	0
48	2	2	0
49	4	33	0
50	5	7	0
51	10	38	0
52	11	43	0
53	12	12	0
54	31	201	0
55	32	198	0
56	61	399	0
57	61	1244	0
58	1	2	1
59	3	1	1
60	3	3	2
61	4	5	2
62	136	132	6

Figure 4.6: Counts of passing and failing executions per test and SPL. t_i is the test id. F_i is the number of failures of t_i. P_i is the number of passing executions of t_i. FC_i is the number of consistent configurations in which t_i fails. We omit test entries without failing runs.

4.3.2.2 Initial Feature Model and Ground Truth

In the experiments, we initialized `SPLif` with an empty feature model, *i.e.*, the execution of `SPLat` at line 12 had no inconsistent configuration. To classify configurations (and hence model the user), we used pre-existing feature models as reference to label consistency of configurations.

4.3.3 Ranking Tests Using Suspiciousness Score

The purpose of test ranking is to speed up discovery of bugs through the prioritization of tests that fail on consistent configurations. Figure 4.7 shows, for each SPL, the ranking of tests from Figure 4.6. Column *R* shows the rank of test *Ti*. The *suspiciousness score* S_i for that test is computed by: $S_i = F_i/(F_i + P_i)$.

Recall that Section 4.2.2 states that the ranking of tests is obtained by lexicographically sorting the triples $\langle FC_i,\ S_i,\ F_i\rangle$. However, at this stage no configurations are known to be (il)consistent. For this reason, we used the pair $\langle S_i,\ F_i\rangle$ and reported only S_i.

Companies

R	t_i	*S*
1	15	0,75
2	16	0,75
3	19	0,75
4	14	0,75
5	18	0,58
6	13	0,50
7	17	0,50

GPL

R	t_i	*S*
1	24	0,97
2	22	0,88
3	21	0,75
4	18	0,50
5	19	0,50
6	25	0,50
7	23	0,50
8	20	0,30

Notepad

R	t_i	*S*
1	31	0,59
2	29	0,52
3	30	0,42

DesktopSearcher

R	t_i	*S*
1	24	1,00
2	25	1,00
3	39	1,00
4	35	1,00
5	36	1,00
6	42	0,94
7	40	0,93
8	43	0,90
9	41	0,88
10	37	0,88
11	38	0,88
12	32	0,75
13	33	0,75
14	34	0,75
15	29	0,60
16	13	0,33
17	27	0,27
18	28	0,27
19	30	0,27
20	10	0,20
21	26	0,20
22	21	0,20
23	11	0,11
24	12	0,11
25	14	0,11
26	15	0,11
27	16	0,11
28	17	0,11
29	19	0,11
30	22	0,11
31	31	0,04
32	18	0,03
33	20	0,03
34	23	0,03

ZipMe

R	t_i	*S*
1	59	0,75
2	62	0,51
3	30	0,50
4	31	0,50
5	60	0,50
6	42	0,50
7	53	0,50
8	48	0,50
9	61	0,44
10	50	0,42
11	58	0,33
12	35	0,33
13	41	0,33
14	32	0,25
15	33	0,25
16	36	0,25
17	37	0,25
18	38	0,25
19	39	0,25
20	40	0,25
21	47	0,25
22	43	0,25
23	44	0,25
24	51	0,21
25	52	0,20
26	46	0,20
27	45	0,14
28	55	0,14
29	54	0,13
30	56	0,13
31	34	0,11
32	49	0,11
33	57	0,05

Figure 4.7: Ranking of tests. Column *R* shows the rank of test t_i from Figure 4.6; *S* shows the suspiciousness score of t_i. A row in gray color indicates a test for which at least one failing configuration it reaches is consistent.

According to Figure 4.7, we observed that for the cases with a relatively small number of failing tests (`Companies`, `GPL`, and `Notepad`) the classification of tests is not very helpful.

Although the first tests with consistent configuration appear high in the ranking (positions 3, 1, and 1, respectively), overall other tests that do not fail on consistent configurations are also highly ranked. For the cases with relatively many failing tests (`DesktopSearcher` and `ZipMe`), we found that ranking tests was helpful: most tests that fail on consistent configurations appear at top positions in the ranking.

RQ1. Based on these results we conclude the following:

The use of a suspiciousness score based on pass-fail ratios of test runs is a good predictor for labeling tests that fail on consistent configurations when the number of failures is relatively high.

4.3.4 Ranking Configurations

Once a test is selected, the tester needs to identify the configurations which are more likely to be consistent out of those that trigger a failure. Such task can be overwhelming: a test run can expose many distinct failures and there could be other tests for inspection. We considered 4 prioritization techniques for inspecting configurations, defined as follows.

- ***Random*** is our comparison baseline. It randomly orders failing configurations associated to a test.
- ***Memory*** is a variation of *Random* that memoizes previously labeled configurations.
- ***Weighted*** is a variation of *Memory* that ranks the configurations for inspection according to their weights.
- ***Adaptive*** is a variant of *Weighted* that re-ranks tests on-the-fly.

We obtain these techniques by setting the option flags (`STATIC`, `MEMORY`, `WEIGHTED`, and `DYNAMIC`) in the `SPLif` algorithm.

When the tester finds a legal configuration for a failing test, `SPLif` skips to the next test, and when he/she finishes inspecting all tests, he/she repairs the test/code for all recognized bugs, *i.e.*, for failures of pairs of consistent configuration and test. Section 4.3.5 discusses another scenario where the tester stops to repair the test/code after finding a failure of a pair of legal configuration and test.

The Figure 4.8 presents the results for these four techniques. It shows the progress in number of consistent configurations found with the progress in number of configurations inspections needed to find the first legal configuration for each test. A datapoint in the plot marks an inspected test. The quicker the plot gets to the top the better; it means more tests are labeled quicker. The narrower the plot the better; it means less configurations require inspection.

The Figure 4.9 summarizes these numbers of configuration inspections and compares them to two baselines: the number of inspections considering all failures (column *Reduction*

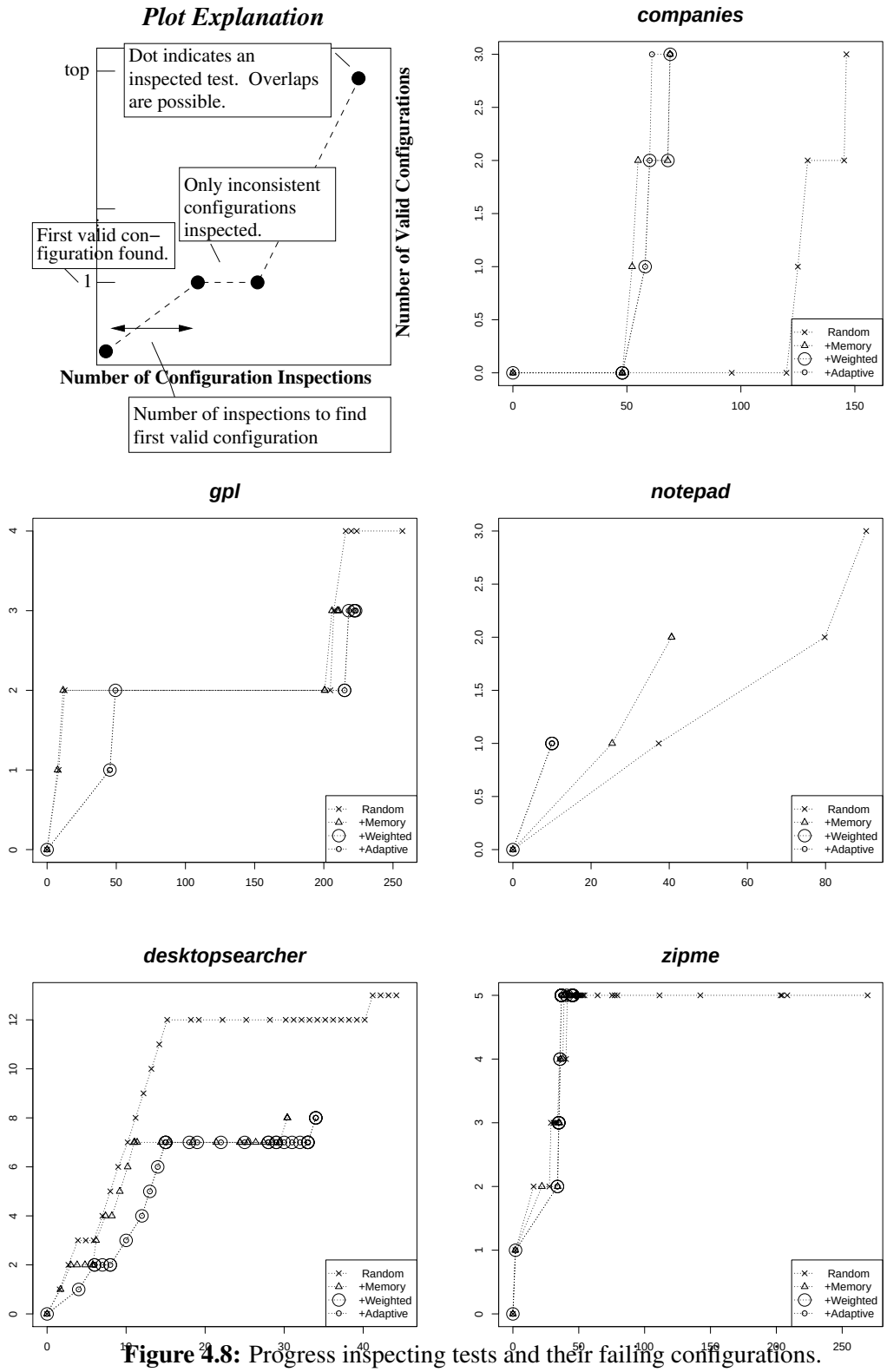

Figure 4.8: Progress inspecting tests and their failing configurations.

All(%), and the number of inspections considering the *Random* technique (column *Reduction Random(%)*. Considering all failures (the rows *All*), the number of inspections is higher than the *Random* mode. In *Random* mode, this number is higher than the remaining modes. In the other modes, the numbers are smaller because `SPLif` recognizes already inspected configurations.

If they appear again, `SPLif` skips to the next test. Thus, `SPLif` is useful to speed up the bug discovery. Using the technique *Memory* it is possible to achieve a reduction from 17.9% up to 88.9% in the number if inspections compared to *Random* mode.

Mode	*#Inspections*	*Reduction All (%)*	*Reduction Random (%)*
Companies			
All	156	-	-
Random	146	6.4%	-
Memory	69	55.8%	52.7%
Weighted and *Adaptive*	69	55.8%	52.7%
DesktopSearcher			
All	151	-	-
Random	44	70.8%	-
Memory	30	80.1%	31.8%
Weighted and *Adaptive*	34	77.5%	22.7%
GPL			
All	770	-	-
Random	257	66.6%	-
Memory	211	72.6%	17.9%
Weighted and *Adaptive*	223	71%	13.2%
Notepad			
All	300	-	-
Random	90	70%	-
Memory	40	86.7%	55.6%
Weighted and *Adaptive*	10	96.7%	88.9%
ZipMe			
All	397	-	-
Random	269	32.2%	-
Memory	45	88.6%	83.3%
Weighted and *Adaptive*	49	87.6%	81.8%

Figure 4.9: The total number of inspections for various techniques, and their reduction compared to two baselines: the number of inspections considering all failures (column *Reduction All(%)*, and the number of inspections considering the *Random* technique (column *Reduction Random(%)*

From the results, we recommend two techniques be used to inspect configurations, depending on the scenario the tester may have:

1. *Memory* is more appropriate when there is a considerable number of tests failing on inconsistent configurations that are common among them; and

2. *Weighted* and *Adaptive* are more appropriate when there is a considerable number of tests failing on consistent configurations that are common among them.

4.3.4.1 Discussion

Results indicate that for `DesktopSearcher`, `GPL` and `Notepad` the use of variant *Random* revealed more consistent configurations than any other variant. Recall that *Random* only uses `SPLif` to rank tests. This is expected as *Random* does not memoize already-labeled configurations, increasing the number of inspections (of consistent configurations or not). The other variants record already-labeled configurations (to detect bugs in other failing tests faster) and hence may not visit all failing configurations. However, for all subjects, the total number of

configuration inspections necessary to inspect all tests was much higher in *Random* compared to other variants. This highlights the importance of `SPLif` to filter out tests that fail on consistent reachable configurations.

Memory is the simplest variant of `SPLif` that ranks both tests *and* configurations. In this variant, `SPLif` memoizes configuration labelings in symbolic models. Intuitively, this enables faster classification of other buggy tests that fail for consistent configurations and helps to ignore failures on inconsistent configurations (lines 27–32 in Figure 4.2). We found that this variant helps to speed up classification of buggy tests when a large number of configurations repeat across tests.

Weighted and *Adaptive* also use memory (*Memory*) of previous labelings. Results indicate that *Memory* performs better compared to these variants for most of the subjects. The reason for this can be attributed to the observation that when there is a considerable number of tests failing on common inconsistent configurations, then it helps if these inconsistent configurations are identified soon. Such a characteristic appears in `ZipMe` (27 tests failing on inconsistent configurations vs. just 5 tests failing on consistent configurations), in `DesktopSearcher` (30 tests failing on inconsistent configurations vs. 13 tests failing on consistent configurations), and in `Companies`. The large number of common inconsistent configurations bring about the difference for `GPL` (see Figure 4.7).

In contrast, for `Notepad`, all failing tests have failures due to consistent configurations. Furthermore, there is 1 consistent configuration common amongst all failing tests. The *Weighted* (which performs heuristic ranking of configurations based on the configurations previously labeled) and *Adaptive* (which re-ranks tests after every labeling) variants help detect this common consistent configuration in just 10 configuration inspections. This leads to the detection of the other buggy tests without requiring any further inspection of configurations. *Memory*, which randomly ranks configurations brings to the top a different consistent configuration (unique to a single test), and then ranks the consistent configuration common to the other tests after 40 inspections.

RQ2. Based on these results we conclude the following:

> The overall number of inspections needed to find problems in tests or source code is fewer when `SPLif` uses an inferred feature model (*Memory*) as opposed to randomly selecting configurations (*Random*). Re-ranking of tests (*Adaptive*) and ranking of configurations (*Weighted*) seem to help when the number of tests failing on common consistent configurations is high.

4.3.5 Incremental Runs of SPLif

Re-execution of `SPLif` for one test (or the entire test suite) after every repair in test (or code) can be expensive, and potentially wasteful if the repair had corrected all faults in the inspected test and the change did not impact any other test. One approach to deal with this cost

is to optimistically assume that the repair in the test corrects all its related faults and, in such a case, not re-execute SPLif. Instead, this approach removes the repaired test from the list and control proceeds to the next test in the ranking. The rationale for this approach is to inspect tests quicker looking for new faults.

After all failing tests have been inspected once, SPLif is *re-started* to run on the new version of the repaired tests alone. Note that all configurations for the other tests must have been inspected in previous runs of SPLif. This incremental pass acts as a validation phase for the test repairs and also helps to identify other faults that could have been exposed by other consistent configurations. This is exactly what we did in our experiments. The plot in Figure 4.8 is for the first pass. We elaborate on the subsequent passes in the following. We observed that, except for GPL, none of the repairs modified the application code.

For all subjects, except DesktopSearcher, SPLif did not reveal any other faults in the second pass. For DesktopSearcher, the first pass discovered 13 tests failing on consistent configurations (out of 34 reported in SPLif's ranking). The second pass found 6 of the 13 repaired tests to fail again. These tests failed for unseen consistent configurations that had not been explored earlier. The third pass again revealed a fault in one of the 6 tests for another unseen configuration.

4.3.6 Discussion of Test Failures

This section discusses the cases where the use of SPLif guided the user to make repairs. Recall that a failure on a consistent configuration indicates a problem in test or code.

4.3.6.1 Companies

Only three tests failed for consistent configurations in this subject. Test 19 displays a window for a company having 2 departments under it, each with employees with respective salaries. The test run execution depresses the cut button in the GUI in order to reduce the cumulative salary of the company and asserts if the cumulative value displayed has been halved. However, when the PRECEDENCE feature is set, it enforces certain constraints on how the salaries of the employees in the company can be updated and thus the cumulative salary is not halved, which results in test failure. This represents a consistent bug in the test code which was too restrictive, the test code was updated to guard the assertion with a check on the PRECEDENCE feature being set or not. Tests 18 and 17 fail for a similar reason; they both require guards based on the use of the PRECEDENCE feature.

4.3.6.2 DesktopSearcher

This subject provides a visual interface where often related components are enabled with the setting of corresponding feature variables. We observed that even a small change in configuration causes a significant change in the interface. For instance, test 42 makes a specific

assertion on the number of files that have been indexed. This test passes for a configuration where only the features SINGLE_DIR, NORMAL_VIEW and TXT are enabled while the rest are disabled. However, when the features HTML and/or LATEX are also enabled, it leads to failures. This happens because the interface changes as it now includes HTML and/or LATEX files in addition to TXT files, which in turn changes the number of indexed files. This was repaired by using different values for the number of indexed files in the assertions conditional to the features being enabled.

All the other failures were similar to test 42; occurring due to the assertions being specific to the respective default configuration settings and were repaired by making them more general, conditioned to the features being enabled. Since the tests are very specific to certain configurations, there are many consistent configurations that expose failures in more than one test or are common amongst the tests (refer Figure 4.5). Hence memory of previous labelings significantly reduces the number of inspections required to detect all buggy tests. Owing to the cross-feature constraints, most of the configurations that the tests fail on are inconsistent configurations. This explains why *Memory* mode does better than *Weighted* and *Adaptive* for this subject.

4.3.6.3 GPL

As indicated in Table 4.7, SPLif ranked 2 of the 4 tests that fail on consistent configurations at the top of the list. The test with the highest ranking (test 24) was the first one to be detected to fail on a consistent configuration. This test invokes the display method of the vertex class. It explicitly sets the WEIGHTED feature variable to true and checks if the appropriate message has been displayed. However, inspection revealed that the message used in the assertion check was inadequate when other features (*e.g.*, CONNECTED and CYCLE) are enabled, leading to assertion errors. This represents a case of a bug in the test code and was detected within 9 inspections using SPLif in *Memory* mode. The next test for which a consistent configuration was detected was for test 22. The test code constructs a graph with 3 vertices and checks if it is strongly connected. However, it fails to assign weights to the edges of the graph. When executed with the WEIGHTED feature set to true, the test throws an IndexOutOfBoundsException when the weights of the edges are accessed.

The bug was corrected in the test by assigning weights to the edges when the WEIGHTED feature is enabled. The consistent configuration detected for test 24 is in common with test 25, which is recorded in *Consistent* (in the *Memory*, *Weighted*, and *Adaptive* modes). Hence, test 25 is detected to be erroneous without requiring additional user inspections to label the configuration as consistent (refer to line 24 of Figure 4.2. The test code contained an assertion that did not behave correctly when the SEARCH feature was set to true. The repair was to modify it to a conditional assertion that restricted its behavior.

Test 23 has a subtle error which gets exposed only on certain configurations, leading to a low suspiciousness score. It tests the method Graph.run that exercises different functionalities

of a graph such as calculation of the shortest path, checking for cycles, calculation of spanning trees using Kruskal and Prim algorithms so on. It conditionally invokes the relevant methods based on the values of the respective feature variables. The Kruskal and Prim algorithms access the weights of the edges and throw an `IndexOutOfBoundsException` when the `WEIGHTED` feature is enabled but weights have not been set. This is an example of a bug in the code of the run method, wherein the calls to methods for Kruskal and Prim, in addition to being guarded by a check on whether the `MSTKRUSKAL` and `MSTPRIM` feature variables are set to true, should be guarded by a condition that checks that when the `WEIGHTED` feature is enabled, the weights of the edges have been set appropriately. The method should return without executing the algorithms otherwise. On resolving this unexpected uncaught exception, the same configuration exposed another error in the test. The assertions were not correct when the spanning tree algorithms were not executed. They were repaired by making the assertions conditional to the `MSTKRUSKAL`, `MSTPRIM` and `WEIGHTED` features being set.

4.3.6.4 Notepad

As illustrated in Table 4.7, `Notepad` has only 3 tests failing for 6 consistent configurations, in total. All these 6 configurations have the feature `UNDOREDOTOOLBAR` set to true, which result in the addition of buttons Undo and Redo to the tool bar. These buttons are disabled in their creation, causing their *toolTipText* attribute to be null. The 3 failing tests use the same code to find a specific button, by iterating over all buttons until finding the one that matches with a given *toolTipText*. All three tests pass through the Undo and Redo buttons on their search path, which leads to `NullPointerException`. In sum, all the tests from `Notepad` fail for the same reason. There is a single configuration that exposes this fault in all 3 tests, which is detected as the first consistent configuration in the *Weighted* and *Adaptive* modes.

The repair is to check whether the *toolTipText* attribute of the buttons in the search path are null, and if so, making the test skip to the next button until finding the searched one. We elaborate on test *testEditToolBar* (see Figure 4.1) as it topped the ranking. When this test runs with a configuration that has the feature `UNDOREDOTOOLBAR` enabled, a null pointer exception is thrown when the *isMatching* method accesses the `REDO` button.

4.3.6.5 ZipMe

Test 59 is the first test in the rank to be analyzed. It checks the functionality of the deflation operation of an input array. After invoking the `deflate` method on an array, it checks if method `finished` returns true, which indicates that the output stream has ended and no more additional output bytes are available. However, when the feature `DERIVATIVE_COMPRESS_ADLER32CHKSUM` is enabled, additional checksum (CRC) bytes are added after the compression. This checks consistency of the assertion on the `finished`

method. We repaired the test by making the assertion conditional on the above mentioned checksum feature.

Test 62 invokes compression and deflation algorithms on zip and gz files, performs file updates, and validates the computed checksum. The failures occur when the checksum feature is not enabled, which was corrected by enforcing that the `ADLER32CHECKSUM` feature is set at the beginning of the test.

Test 60 invokes a method to compute the `CRC32` data checksum of a compressed data stream. It then ensures that the zipped file is not yet available by asserting that the return value is -1. However, when this test is performed with the `EXTRACT` feature being enabled, the return value is 1 or 0 (not -1), depending on whether the zip file has been extracted fully or not. We repaired the assertion on this test to consider the variation due to `EXTRACT`.

Test 61 is a similar test that explicitly updates the checksum of a zip file and then adds a new zip entry. It asserts that the CRC of the new zip entry is 0. However, when `DERIVATIVE_COMPRESS_CRC` is enabled, the previous CRC is passed onto the new entry, which leads to assertion failure. The bug which was corrected by adding a conditional assertion.

Failure for test 58 was also a bug in the test code resolved by adding a conditional assertion on the `ARCHIVE_CHECK` feature.

4.3.7 Threats to Validity

As usual one threat to external validity is that our case studies may not have be representative of all programs. To mitigate this threat we considered not only open-source previously-used SPLs but also one real configurable system (GCC, see Chapter 5) that has been under active development for almost three decades and is tested by a wide community.

The selection of tests is another threat to generalization, the tests may be insufficient or inappropriate. We tried to reduce this threat by using tests requirements (Instructions to students on generating new tests for SPLs., 2014) to ensure good coverages of: code, functionality, and feature..

Other threat is the potential high number of variables in incomplete configurations that testers may need to inspect. To mitigate this problem we ran our techniques on large code and observed that many variables appear undefined in several configurations to inspect, confirming our expectations (as discussed in the previous chapter) that not all variables are accessed in every path. It is also important to note that `SPLif` favors configurations with more undefined variables for inspection (see "$?_c$" in Section 4.2.3).

Finally, another threat to generalization is the assumption that the labeling provided by the user is accurate. `SPLif` allows the user to skip the labeling of a configuration that he/she is not sure of. This problem could also be mitigated by checking the consistency of each labeling with the feature model learned until that point.

5

Case Study: GCC

In this chapter we describe a case study conduced with the aim of illustrating `SPLat` and `SPLif` with the GNU Compiler Collection (GCC) (GCC, 2014), a large system with more than 7 million KLOCs, more than 17k tests, with hundreds of configuration options (GCC Options, 2014), 2,015 features. It has been developed for almost three decades by over 500 contributors.

First, we introduce the goals and the research questions. Next, we describe the general infrastructure. And finally, we discuss the results.

5.1 Research Questions

Our main goals are twofold: (1) to provide better support for testing a large configurable system and what kind of failures `SPLif` could find; (2) to observe how `SPLat` and `SPLif` could scale to such a large system. Based on those goals we pose the following research questions:

RQ1 How does `SPLif` rank faulty tests and configurations (of selected tests) for inspection?

RQ2 How does `SPLat` and `SPLif` scale to real code?

With the recent work on testing GCC (YANG et al., 2011; CHEN et al., 2013; LE; AFSHARI; SU, 2014), we did not expect to find new bugs. However, the use of `SPLif` did reveal new bugs. Since we do not have extensive GCC knowledge to properly classify general failures, we focused our inspection of failures on crashes, which provide a stronger indication of a real bug in code. Other kinds of failures although also interesting typically manifest because of overly specific assertions (*i.e.*, fragile tests). In fact, the crashes we observed occurred prior to the execution of the test assertions.

5.2 General Infrastructure

5.2.1 The GCC Testing Infrastructure

The GCC testing infrastructure runs each test for only one configuration; we used `SPLat` to run each test for multiple configurations, reachable from tests, and then we applied `SPLif` to rank both the failing tests and configurations, as we did in Section 4.3.

We briefly introduce DejaGnu (DejaGnu, 2014), the GCC testing framework. The code snippet in Figure 5.1 shows an example DejaGnu test:

```
1  /*{ dg-do compile }*/
2  /*{ dg-options "-std=gnu89 -Wformat"}*/
3  #include format.h
4  void foo(char **sp, wchar_t **lsp) { ... }
```

Figure 5.1: Example of GCC test using DejaGnu.

This test is for the C front-end of GCC. The directive `dg-do` instructs DejaGnu to only compile the function `foo`. Other directives can run other tasks (*e.g.*, preprocess, assemble, link, and run) on this test and combine these tasks. The directive `dg-options` instructs DejaGnu to override the default option values with the assignments that follow. In this example, the code will be compiled according to two options: "`-std=gnu89`" (uses ANSI C dialect) and "`-Wformat`"(checks format of string arguments of several functions). DejaGnu determines test verdicts by matching specified regular expressions with the outputs of test runs.

5.2.2 Implementation

The two versions of `SPLat` (for both languages Java and Ruby on Rails) were not able to run GCC code, because (1) these two versions of `SPLat` were written in the same language of the subject programs used, therefore they do not work for GCC, that is written in the C language; (2) these versions only consider boolean values (*true/false* or *0/1*) for the feature variables, and in GCC they may have multiple values as we will see forward.

In order to avoid developing another version of `SPLat` in C language, we decided to monitor the feature variables (exercised during the test execution) using log files containing the values that feature variables assume during execution. This Resulted on a new version of `SPLat` for C and probably other languages, developed in Java, that supports those files and feature variables with multiple values.

5.2.2.1 Instrumentation

It is important to distinguish *input option* from *feature variable*. Input options correspond to the configuration parameters passed to the system, typically on the command line (*e.g.*,

"`-O1`"). Feature variables are the program variables that reflect these options in code (*e.g.*, "`optimize`").

To enable execution of `SPLat` on GCC (see Figure 4.2, line 12), we instrumented GCC to monitor all accesses to feature variables by replacing every feature variable access by a function that logs both the name and value of the feature, and return the feature value for the code, as expected by the original code. The execution of each test on instrumented GCC produces a log file containing both the name and the values that feature variables assume during execution. We did *not* model all options (see Section 5.3.3) but we did monitor all feature variables. It is important to note that this process can be automated, by using a tool for that, such as Pin[1] a dynamic binary instrumentation tool.

5.2.2.2 Execution

Considering the execution of the test code in Figure 5.2, it was produced a log file illustrated in Figure 5.3a with the feature variables exercised and corresponding values observed during the execution. Note that not all feature variables and values were reported due to the great amount of lines in the log file.

```
1  /* PR debug/47684 */
2  /* {dg-do compile} */
3  /* {dg-options "-O3 -fcompare-debug"} */
4  /* {dg-xfail-if "" {powerpc-ibm-aix*} {"*"} {""} } */
5
6  int in[8][4];
7  int out[4];
8
9  void
10 foo (void)
11 {
12   int sum = 1;
13   int i, j, k;
14   for (k = 0; k < 4; k++)
15     {
16       for (j = 0; j < 4; j++)
17         for (i = 0; i < 4; i++)
18           sum *= in[i + k][j];
19       out[k] = sum;
20     }
21 }
```

Figure 5.2: Test pr47684.c from gcc.dg test suite.

To determine new input vectors from variable accesses we manually constructed a map to express the correspondence between each feature (and respective values) and options, Figure 5.3b shows a sample of this map. Although the construction of such map can be automated (XU et al., 2013), the GCC input options can enable (or disable) multiple feature variables (not only boolean values), and those relationships (feature - options - values) are defined in both the compiler source code and special option definition files, making it difficult to extract automatically.

For instance, the feature variable "`optimize`" can assume one of these four values: "0", "1", "2" or "3" inside the GCC code. The map (in Figure 5.3b) expresses the correspondence

[1] `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`

[splat] pedantic : 0
[splat] align_loops : 1
[splat] align_jumps : 1
[splat] align_functions : 1
[splat] optimize : 3
[splat] flag_unroll_loops : 0
[splat] flag_pic : 0
[splat] flag_asan : 0
[splat] flag_tree_vectorize : 0
...

(a) Log file.

Feature Vars.	*Values*	*Options*
flag_asan	0, 1	−fno−sanitize=address −fsanitize=address
optimize	0, 1, 2, 3	−O0, −O1, −O2, −O3
flag_unroll_loops	0, 1	−fno−unroll−loops, −funroll−loops
flag_ipa_cp	0, 1	−fno−ipa−cp, −fipa−cp
flag_tree_vectorize	0, 1	−fno−tree−vectorize, −ftree−vectorize
flag_schedule_insns	0, 1	−fno−schedule−insns, −fschedule−insn
...	...	...

(b) A sample of the feature-value-option map.

−O3/−fno−unroll−loops/−fno−sanitize=address/−fno−tree−vectorize
−O3/−fno−unroll−loops/−fno−sanitize=address/−ftree−vectorize
−O3/−fno−unroll−loops/−fsanitize=address/−fno−tree−vectorize
−O3/−fno−unroll−loops/−fsanitize=address/−ftree−vectorize
−O3/−funroll−loops/−fno−sanitize=address/−fno−tree−vectorize
−O3/−funroll−loops/−fno−sanitize=address/−ftree−vectorize
−O3/−funroll−loops/−fsanitize=address/−fno−tree−vectorize
−O3/−funroll−loops/−fsanitize=address/−ftree−vectorize

(c) Options generated.

Figure 5.3: Artifacts used and produced during `SPLat` execution.

between each feature (and respective values) and options: `optimize=0` with the option `-O0`, `optimize=1` with the option `-O1`, `optimize=2` with the option `-O2`, and `optimize=3` with the option `-O3`.

From the log file (in Figure 5.3a) of accessed variables produced with the test run (in Figure 5.2) and the variable-value-option map (in Figure 5.3b), `SPLat` is able to generate new test inputs (*i.e.*, option vectors). `SPLat` starts execution with the configuration obtained from the options declared with the directive `dg-options` (line 3); feature variables not related to referred options assume default values.

Since we did not model all feature variables in our map (*c.f.*, Section 5.3.3), `SPLat` only explores a subset of those observed during the execution (see the log file in Figure 5.3a) that were modeled in the map (Figure 5.3b), resulting on these four feature variables and options: `optimize=3` `flag_unroll_loops=0`, `flag_asan=0`, and `flag_tree_vectorize=0`. From them, `SPLat` can explore up to eight option vectors as showed in Figure 5.3c, this is the maximum number of vectors (combinations) for this exam-

ple, because the option `-O3` is constant due to the line 3 in Figure 5.2, and if one consider an initial feature model, this amount may be lower.

5.3 Setup

5.3.1 Tests Execution

Test execution in GCC is time-consuming. For example, it takes ∼45min to run all 2,608 tests from the `dg-gcc` test suite considering 1 configuration per test and our running environment. This corresponds to roughly 1s per test run. To deal with the high cost of test execution, we focused on a selection of test suites, ran `SPLat` using a limited number of options, limited the number of configurations per test to 50, and randomized the execution of `SPLat` to sample different configurations, but reachable from the test runs.

5.3.2 Tests Analyzed

Overall, we analyzed a total of 4,108 tests from three different test suites: 2,608 from `gcc-dg`, 548 from `dg-torture`, and 952 from `tree-ssa`. We focused on these suites for this experiment because we observed from bug-reports that the incidence of bugs revealed with these suites was higher.

5.3.3 Options Analyzed

To make the runtime reasonable, we restricted the number of options that `SPLat` considers (see GCC Options (2014)). From a total of 400 most frequently cited options in the GCC bug reports from the month of July 2014, we used the 40 (10%) most frequent. The rationale was that more bugs could be found close to where some bugs have been recently reported (we noticed that the top options do not change much across different months).

5.3.4 Initial Feature Model and Ground Truth

In this experiment we initialized `SPLif` with an empty feature model (*i.e.*, the execution of `SPLat` at line 12 is unconstrained) and used an existing feature model of GCC (GARVIN; COHEN; DWYER, 2013) as the ground truth to model developer knowledge during for classifying configurations. Other choices of initial model and ground truth are possible.

For the ground truth we built on the work of Garvin *et al.* (GARVIN; COHEN; DWYER, 2013) that documented 110 constraints from GCC. We augmented this model with constraints that we manually extracted from the online documentation of GCC (GCC Documentation, 2014); we found a total of 136 new constraints. For example, we found that the option *-fsel-sched-pipelining* enables software pipelining of the innermost loops during selective scheduling and

has no effect unless the options *-fselective-scheduling* or *-fselective-scheduling2* are turned on. We modeled these constraints using the Z3 as a SAT solver (Z3 THEOREM PROVER, 2014), because it has a function of *unsat-core*, that will be used to discover new constraints in Section 5.4.3.

5.4 Results

5.4.1 Ranking Tests and Configurations

t_i	F_i	P_i	FC_i
...			
4028	7	43	0
4029	13	36	0
4030	1	47	1
4031	2	44	1
4032	6	44	3
4033	6	43	4
4034	7	33	3
4035	7	43	4
4036	8	20	4
4037	9	41	4
4038	9	39	5
4039	10	28	2
4040	10	40	3
4041	10	40	4
4042	10	40	10
4043	11	36	8
4044	12	27	6
4045	12	34	7
4046	12	38	11
4047	12	38	12
4048	12	38	12
4049	13	37	13
4050	13	37	13
4051	14	36	2
4052	14	36	5
4053	14	35	6
4054	14	36	6
4055	16	34	4
4056	16	34	6
4057	16	34	7
4058	16	34	7
4059	17	33	6
4060	18	32	3
4061	18	32	13
4062	20	27	6
4063	20	30	12
4064	21	20	3
4065	21	29	8
4066	22	26	5
4067	22	28	8
4068	25	25	7
4069	25	0	12
4070	27	23	12

(a) Passing and crashing failure executions per test.

GCC		
R	t_i	S
1	4069	1,00
2	4070	0,54
3	4064	0,51
4	4068	0,50
5	4066	0,46
6	4067	0,44
7	4062	0,43
8	4065	0,42
9	4063	0,40
10	4060	0,36
11	4061	0,36
12	4059	0,34
13	4055	0,32
14	4056	0,32
15	4057	0,32
16	4058	0,32
17	4044	0,31
18	4036	0,29
19	4053	0,29
20	4051	0,28
21	4052	0,28
22	4054	0,28
23	4029	0,27
24	4039	0,26
25	4045	0,26
26	4049	0,26
27	4050	0,26
28	4046	0,24
29	4047	0,24
30	4048	0,24
31	4043	0,23
32	4040	0,20
33	4041	0,20
34	4042	0,20
35	4038	0,19
36	4037	0,18
37	4034	0,18
38	4028	0,14
39	4035	0,14
40	4033	0,12
41	4032	0,12
42	4031	0,04
43	4030	0,02

(b) Ranking of tests.

Figure 5.4: Statistics on Tests. In the left table, column R shows the rank of test t_i from Figure 5.4a; S shows the suspiciousness score of t_i. A row in gray color indicates a test that requires inspection; a test for which at least one failing configuration it reaches is consistent. In the right table, t_i is the test id; F_i is the number of crashing failures of t_i. P_i is the number of passing executions of t_i; FC_i is the number of consistent configurations in which t_i crashes. We omit test entries without crashing runs.

The main goal of ranking both tests and configurations is to get a consistent configuration that fails quickly. In the following we present some statistics related to the ranking of tests:

- Recall that we focused only on crash failures. From the total of 4,108 tests that we analyzed, 497 tests failed (either due to crash or not). In total, 3,986 pairs of tests and configurations failed (either due to crash or not). Recall that we ran each test against 50 reachable configurations;
- Considering only crashes, a total of 43 tests manifested crashes ($\sim$8.65% of the total number of failing tests and $<$2% of the total number of tests) in 268 pairs of test and configurations ($\sim$6.73% of the total number of failing pairs);
- According to Figure 5.4, from the total of 43 crashing tests, only 2 tests had all crashing configurations inconsistent. These bad cases ranked lower, at positions 23 and 38 positions.

Figure 5.5 illustrates the progress in the number of consistent configurations found with the progress in number of configurations inspected, considering the four alternatives modes presented in Section 4.3.4. From a total of 268 failing configurations (218 distinct), only 49 configurations needed inspection (26 of which were consistent), considering *Weighted* (or *Adaptive*) mode. This indicates that with a relative low number of configuration inspections `SPLif` is enabled to find a consistent failing configuration in all 43 crash revealing tests.

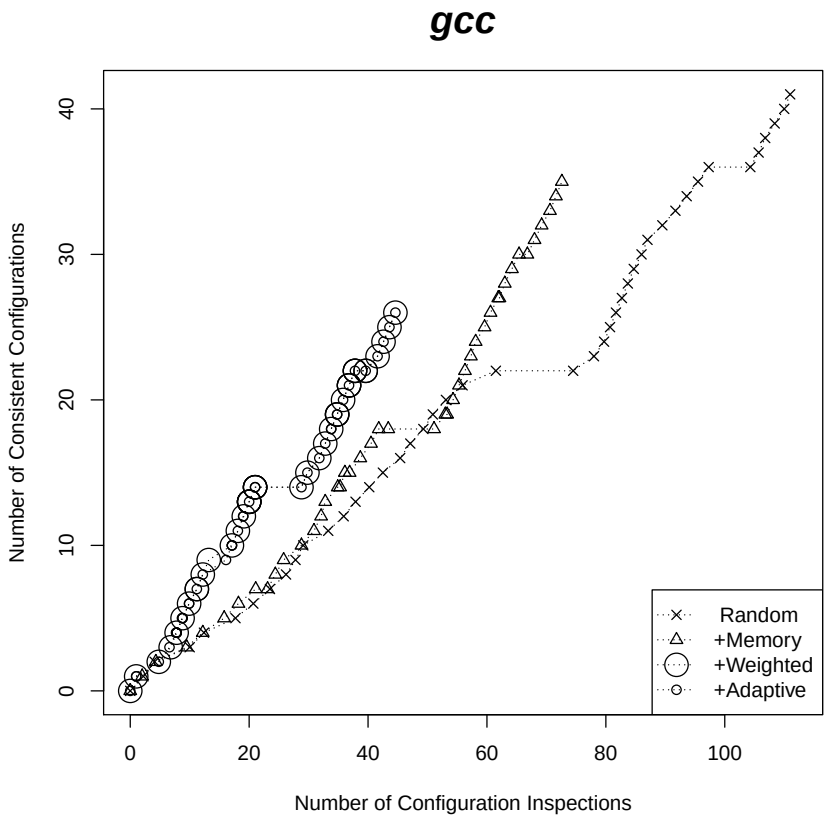

Figure 5.5: Configuration inspection progress for GCC.

The Figure 5.6 summarizes these numbers of configuration inspections and compares them to two baselines: the number of inspections considering all failures (column *Reduction All(%)*, and the number of inspections considering the *Random* technique (column *Reduction Random(%)*. This result represents a reduction of 55.8% compared to random inspection, and a reduction of 58.6% compared to inspect all 268 failing configurations.

GCC			
Mode	*#Inspections*	*Reduction All (%)*	*Reduction Random (%)*
All	268	-	-
Random	111	58.6%	-
Memory	81	69.7%	27%
Weighted and *Adaptive*	49	81.7%	55.8%

Figure 5.6: The total number of inspections for various techniques, and their reduction compared to two baselines: the number of inspections considering all failures (column *Reduction All(%)*, and the number of inspections considering the *Random* technique (column *Reduction Random(%)*

RQ1. Based on these results we conclude the following:

`SPLif` ranked tests that fail in consistent configurations in higher positions. The overall number of inspections required to identify bugs is much fewer when `SPLif` uses *Adaptive* or *Weighted* modes to rank configurations, because the number of tests failing on common consistent configurations is higher than the number of tests failing on common inconsistent configurations. From a total of 49 configurations inspected, 26 configurations were labeled as consistent.

5.4.2 New Bugs Found

We found a total of 5 bugs in the source code, distributed over 268 pairs of tests and configurations (see Figure 5.7). To provide informative bug reports to the GCC team we needed to simplify observed failures. For that we grouped failing pairs (of tests and configurations) in clusters and minimized configurations (from each pair) inside clusters. We clustered failing pairs according to the GCC error message and the code location responsible for the failure/crash, when it is available. For example, all crashes with the message "int_cst_value" corresponding to location `tree.c:10625` were grouped in one cluster. Figure 5.7 shows all clusters defined according to these heuristics, and lists the crashes we found in the main trunk of GCC. Column "*Name*" presents the clusters, column "*Tests*" shows the number of crashing tests, column "*Pairs*" lists the number of pairs (test, configuration) that crashes, column "*Id*" denotes the id of the bug as reported in the GCC bug-tracking system, column "*Date Confirmed*" shows the date the team confirmed the bug as new. A bug report is initially given the status "UNCONFIRMED". Column "*Current Status*" shows the current status of the bug, and finally column "*Date Fixed*" shows the date the GCC team fixed the bug.

Unfortunately, there may be configuration options that do not contribute to reproducing

Name	*Id*	*#Tests*	*#Pairs*
`compute_affine_dependence, tree-data-ref.c: 4233`	61980	34	223
`int_cst_value, tree.c: 10625`	62069	4	34
`verify_ssa failed, tree-ssa.c: 1056`	62070	1	6
`build2_stat, tree.c: 4265`	62140	1	4
`Segmentation fault: 11`	62141	1	1

(a) Clusters of crash-revealing failures.

Id	*Date Confirmed*	*Current Status*	*Date Fixed*
61980	Aug.1,2014	RESOLVED FIXED	Jul.18,2015
62069	Aug.8,2014	NEW	-
62070	Aug.8,2014	RESOLVED FIXED	Aug.11,2014
62140	Aug.14,2014	RESOLVED FIXED	Oct.16,2014
62141	Aug.14,2014	RESOLVED FIXED	Nov.19,2014

(b) Bug report ids.

Figure 5.7: GCC bugs. Details at: `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=`. Bug ids are sorted by date the bug was confirmed as new.

the crash. For that reason, we manually applied delta-debugging (ZELLER, 1999) to simplify configurations inside each bucket. More specifically, we selectively removed options from the input configuration and re-executed the test in separate. We repeated this process until the test would no longer reveal the crash.

We applied these simplification mechanisms as follows. For each cluster we randomly picked one pair of test and configuration and minimized the configuration to reproduce the crash at the same location. Then we re-run *all tests* from the same cluster with the minimized configuration to confirm that we would be able to reproduce the crash. The average number of options before minimization was 7. With minimization we found that only 1 or 2 options were relevant to reproduce crashes.

With the minimized configuration at hand, we filed a bug-report, one per cluster. When we observe that the GCC team fixed 3 of the reported bugs (case of bugs: 62070, 62140, 62141) we validated the fix by re-running all pairs of tests and configurations inside the corresponding cluster for the updated version of GCC. If this validation fails it would likely indicate a different problem which would demand the split of the cluster and the report of a new bug. That did not happen for those 3 bugs, *i.e.*, apparently, the fixes were effective. According to the bug reports of the remaining bugs, they have not been fixed because they explores a problematic feature, that should have already been removed, and no one wants to touch this code, apparently.

5.4.3 New Configuration Constraints Found

Although our goal is not to infer feature models, in the process of analyzing failures, we encounter inconsistent configurations, which we documented to accelerate the inspection process.

We used the GCC online documentation (GCC Documentation, 2014) to classify configurations as consistent or not.

For example, we found that the options *-fsched-spec-load* and *-fsched-spec-load-dangerous* allow speculative motion of some load instructions, and can be only applied when scheduling instructions before register allocation, *i.e.*, with *-fschedule-insns* or at *-O2* or higher. We also found that the option *-fsel-sched-pipelining* enables software pipelining of the innermost loops during selective scheduling and has no effect unless *-fselective-scheduling* or *-fselective-scheduling2* is turned on.

Considering both modes *Weighted* and *Adaptive* (see Figure 5.5), we inspected a total of 49 configurations. Out of these, 19 were inconsistent and led to the discovery of 5 distinct constraints. Considering all inspection modes, we found a total of 6 constraints in addition to the Garvin *et al.* (GARVIN; COHEN; DWYER, 2013) model. We used the *unsat-core* function from the SAT solver in order to check if the new constrains were new, and therefore detect the new constraints.

RQ2. Based on these results we found the following:

> Considering the number of inspections to find bugs, our proposed technique helped to quickly reveal new bugs on GCC, a large configurable system that has been under active development for almost three decades.

5.5 Threats to Validity

The selection of tests is one important threat to generalization, for GCC, we used all test suites (hundreds of tests) related to the features we tested. Other important threat is that our selection of feature options to analyze in GCC may introduce bias. It is possible that on a larger set of options more configurations would fail, creating a different scenario for `SPLif`. We remain to evaluate how different heuristics (for selecting feature options) influences `SPLif` results.

Another threat is the selection of 50 configurations, mentioned in Section 5.3.1, where we randomized the execution of `SPLat` to sample different configurations, instead of picking the first fifty explored configurations. However, this sampling can be done in different ways, resulting on different configurations, and creating a different scenario with different failures (more or less). We leave it as a future work to explore a good strategy to sample from these configurations (COHEN; DWYER; SHI, 2006, 2007; MCGREGOR, 2001).

Although our techniques to rank configurations and decrease the number of inspections seem to work well, they could further reduce the amount of inspections. For instance, we only discovered 5 constraints from 19 inconsistent configurations inspected, because the user only labels configurations instead of violated constraints. Consequently, the inferred model ($\mathscr{M}_I$) will only be able to check if a configuration is inconsistent when it is more complete, leading to some

unnecessary inspections. One way to mitigate this threat could be asking for the user to inform the constraint violated by the configuration inspected, adding the constraint to the model instead of the inconsistent configuration.

Finally, the degree of incompleteness of the feature model may impact the results, increasing or decreasing the number of inspections. To alleviate this threat, we used one existing feature model (incomplete) as an initial model to avoid exploration of known inconsistent configurations.

6

Related Work

This chapter discusses the related work regarding to `SPLat` and `SPLif`.

6.1 SPLat

6.1.1 Dynamic Analysis

Korat. `SPLat` was inspired by Korat BOYAPATI; KHURSHID; MARINOV (2002), a test-input generation technique based on Java predicates. Korat instruments accesses to object fields used in the predicate, monitors the accesses to prune the input space of the predicate, and enumerates those inputs for which the predicate returns true. Directly applying Korat to the problem of reducing the combinatorics in testing configurable systems is not feasible because the feature model encodes a *precondition* for running the configurable system, which must be accounted for. In theory, one could automatically translate a (declarative) feature model into an imperative constraint and then execute it before the code under test, but it could lead Korat to explore the *entire* space of feature combinations (up to 2^N combinations for N features) before *every* test execution. In contrast, `SPLat` exploits feature models while retaining the effectiveness of execution-driven pruning by applying it with SAT in tandem. Additionally, `SPLat` can change the configuration being run during the test execution (line 44 in Figure 3.4), which Korat did not do for data structures.

Shared execution. Starting from the work of d'Amorim *et al.* D'AMORIM; LAUTERBURG; MARINOV (2007), there has been considerable ongoing research on saving testing time by sharing computations across similar test executions APEL et al. (2013); AUSTIN; FLANAGAN (2012); CLASSEN et al. (2010); D'AMORIM; LAUTERBURG; MARINOV (2007); HOSEK; CADAR (2013); KäSTNER et al. (2012); KIM; KHURSHID; BATORY (2012); KOLBITSCH et al. (2012); RHEIN; APEL; RAIMONDI (2011); TUCEK; XIONG; ZHOU (2009). The key observation is that repeated executions of a test have much computation in common. For example, Shared Execution KIM; KHURSHID; BATORY (2012) runs a test simultaneously against several SPL configurations. It uses a *set* of configurations to support test execution,

and splits and merges this set according to the different decisions in control-flow made along execution. The execution-sharing techniques for testing SPLs differ from `SPLat` in that they use *stateful* exploration; they require a dedicated runtime for saving and restoring program state and only work on programs with such runtime support. Consequently, they have high runtime overhead not because of engineering issues but because of fundamental challenges in splitting and merging state sets at proper locations. In contrast, `SPLat` uses *stateless* exploration GODEFROID (1997) and never merges control-flow of different executions. Although `SPLat` cannot share computations between executions, it requires minimal runtime support and can be implemented very easily and quickly against almost any runtime system that allows feature variables to be read and set during execution.

Sampling. Sampling exploits domain knowledge to select configurations to test. A tester may choose features for which all combinations must be examined, while for other features, only t-way (most commonly 2-way) interactions are tested COHEN; DWYER; SHI (2006, 2007); MCGREGOR (2001). Our dynamic program analysis *safely* prunes feature combinations, while sampling approaches can miss problematic configurations APEL et al. (2013).

Spectrum of SPL testing techniques. Kästner *et al.* KäSTNER et al. (2012) define a spectrum of SPL testing techniques based on the amount of changes required to support testing. On the one end are black-box techniques that use a conventional runtime system to run the test for each configuration; `NewJVM` is such a technique. On the other end are white-box techniques that extensively change a runtime system to make it SPL-aware; shared execution is such a technique. `SPLat`, which only requires runtime support for reading and writing to feature variables, is a lightweight white-box technique that still provides an optimal reduction in the number of configurations to consider.

6.1.2 Static Analysis

Kim *et al.* KIM; BATORY; KHURSHID (2011) developed a static analysis that performs reachability, data-flow and control-flow checks to determine which features are relevant to the outcome of a test. The analysis enables one to run a test only on (all valid) combinations of these relevant features that satisfy the feature model. `SPLat` is only concerned with reachability, so even if it encounters a feature whose code has no effect, it will still execute the test both with and without the feature. But a large portion of the reduction in configurations in running a test is simply due to the idea that many of the features are not even reachable. Indeed, as Section 3.3 shows, `SPLat` determines reachable configurations with much greater precision and is likely to be considerably faster than the static analysis because `SPLat` discovers the reachable configurations during execution. Static analysis may be faster if its cost can be offset against many tests (because it needs only be run once for one test code that allows different inputs), and if a test run takes a very long time to execute (*e.g.* requiring user interaction). But such situations do not seem to arise often, especially for tests that exercise a small subset of the codebase.

6.2 SPLif

6.2.1 Product Line Testing

Testing software product lines is an active area of research KIM et al. (2013); BORBA et al. (2013); APEL et al. (2013); SHI; COHEN; DWYER (2012); SONG; PORTER; FOSTER (2012); KIM; KHURSHID; BATORY (2012); KäSTNER et al. (2012); KIM; BATORY; KHURSHID (2011); GARVIN; COHEN (2011); CABRAL; COHEN; ROTHERMEL (2010); UZUNCAOVA. (2008); QU; COHEN; ROTHERMEL (2008). The main focus of prior work is to optimize test *execution*. Two approaches have been considered: (1) detection of relevant products to test and (2) optimization of execution of sets of products.

For the first part the focus is to find the set of products that a test must be run against. Kim *et al.* KIM; BATORY; KHURSHID (2011) proposed a static analysis to compute a conservative approximation for the set of *relevant* products to run a given test. SPLat KIM et al. (2013) also computes a sound estimate for the set of relevant products to test but it uses a low-overhead dynamic analysis, specifically execution-driven monitoring BOYAPATI; KHURSHID; MARINOV (2002), to determine relevant products.

For the second part the focus is to reduce the cost of running a test against the products that must be executed. Kim *et al.*'s shared execution KIM; KHURSHID; BATORY (2012) allows sharing the results of computations that are common across different tests, thereby allowing certain results, which under traditional execution would be computed multiple times, to be computed just once. Kästner *et al.* KäSTNER et al. (2012) propose to model variability as non-deterministic choices and then using a model checker to run the tests – which shares computations common to different products and avoids the need to enumerate products for a test. Nguyen *et al.* NGUYEN; KäSTNER; NGUYEN (2014) extends that work by applying the technique proposed by Kästner *et al.* KäSTNER et al. (2012) to applications that build on top of plugins. `SPLif` can benefit from test execution optimization by computing its results more quickly. Moreover, `SPLif`'s output could be used to guide the selection of subsequent configurations for execution.

Qu et. al. QU; COHEN; ROTHERMEL (2008) focus on regression testing of evolving configurable software systems. They present an empirical study about the impact of configuration selection heuristics used in regression testing on fault-detection capability. Their results highlight that a number of bugs may be missed if certain configurations are not tested and that prioritizing configurations allows for more effective testing. It is natural to consider the context of regression testing for applying `SPLif`; this context would allow failing and passing runs across different program versions to be compared and analyzed to more accurately identify causes of test failures and inconsistent configurations.

Uzuncaova's *incremental* approach UZUNCAOVA. (2008) addresses the test input generation problem for product lines. The approach generates input for a product by *augmenting*

previously generated inputs for other products. Test input generation techniques can directly enhance the usefulness of SPLif, by providing it more passing and failing runs to analyze.

Al-Hajjaji *et al.* AL-HAJJAJI et al. (2014) propose a technique to speedup sampling based on configuration dissimilarity. The rationale of this strategy is that dissimilar test cases are likely to detect more defects than similar ones. Although SPLif uses similarity to rank configurations, the goal is different SPLif uses similarity to speedup discovery of consistent failing configurations whereas Al-Hajjaji *et al.* use dissimilarity to speedup sampling. We plan to investigate how dissimilarity can improve SPLif even further.

6.2.2 Feature Model Extraction and Inference

There is a large body of work on inferring/extracting feature models CZARNECKI; WASOWSKI (2007); ALVES et al. (2008); WESTON; CHITCHYAN; RASHID (2009); SHE et al. (2011); RABKIN; KATZ (2011); LOPEZ-HERREJON et al. (2012); ACHER et al. (2012); HASLINGER; LOPEZ-HERREJON; EGYED (2013); DAVRIL et al. (2013); XU et al. (2013) that include: static analysis to extract feature dependencies from code, information retrieval and data mining, evolutionary search, and algorithms based on propositional logic. She *et al.* SHE et al. (2011), Rabkin and Katz RABKIN; KATZ (2011), and Xu *et al.* XU et al. (2013) use static analysis to extract feature dependencies from code. Alves *et al.* ALVES et al. (2008) and Davril *et al.* DAVRIL et al. (2013) use information retrieval/data mining. For example, Davril *et al.* DAVRIL et al. (2013) use text mining to build a product-by-feature matrix, use AI's association rule learning to mine feature associations, and use a mix of specialized algorithms and data mining to build diagrams. They assume there is an input list of *consistent* configurations.

Lopez-Herrejon *et al.* LOPEZ-HERREJON et al. (2012) use evolutionary search. Haslinger *et al.* HASLINGER; LOPEZ-HERREJON; EGYED (2013) provide custom algorithms, they assume that the user provides a list of all consistent configurations. Czarnecki and Wasowski CZARNECKI; WASOWSKI (2007) extract feature models from propositional formulas. Acher *et al.* ACHER et al. (2012) synthesize feature models by merging sets of product descriptions. Weston *et al.* WESTON; CHITCHYAN; RASHID (2009) proposed a guided process to generate feature models, based on natural-language requirements documents, and represented these models in a way which details their semantic composition. None of the above cited works exploit tests and their executions. SPLif uses test failures from inconsistent configurations to infer incomplete feature models *to support testing*, which is intrinsically incomplete. The techniques above can complement SPLif by making initial feature models more complete; hence reducing effort in user inspection.

Recent techniques have been proposed to analyze and validate feature models. Segura *et al.* SEGURA et al. (2010) propose using metamorphic testing for the automated generation of test data for the analyses of feature models. Henard *et al.* HENARD et al. (2013) propose an automated approach to find and fix inconsistencies between system and re-engineered feature

model, such as: (1) system configurations derived from the FM are incorrect with respect to the system, and (2) existing valid configurations do not satisfy the feature model formula. After finding those inconsistencies, they try to automatically fix them, so that the FM reflects its system. Any further development to improve feature models will benefit `SPLif`.

6.2.3 Fault Localization

In this context, suspiciousness metrics have been proposed to rank code entities in terms of their likelihood of containing faults JONES; HARROLD; STASKO (2002); RENIERIS; REISS (2003); DALLMEIER; LINDIG; ZELLER (2005); ABREU; ZOETEWEIJ; GEMUND (2006, 2007, 2009); CAMPOS et al. (2013). For example, Tarantula JONES; HARROLD; STASKO (2002) is a tool that marks a statement as possibly faulty if it is *primarily* executed by failing runs than by passing runs. It associates with each statement a suspiciousness metric that indicates the likelihood of the statement being faulty based on the proportion of failing runs executing it versus passing runs.

The Ochiai metric ABREU; ZOETEWEIJ; GEMUND (2006) is a more recently proposed measure of the suspiciousness of a statement to be faulty, with its roots in biological study. Campos *et al.* CAMPOS et al. (2013) recently proposed the use of information gain (in particular, the entropy in the coverage profiles of test runs) to guide test generation so to obtain better fault localization. In contrast with these approaches, `SPLif` ranks tests based on the ratio of failing and passing configurations instead of ranking statements based on the ratio of failing and passing test runs.

Ghandehari *et al.* GHANDEHARI et al. (2013) generate failure-inducing configurations – for which they run existing tests – in an attempt to better localize faulty statements in code. `SPLif` also discover faulty configurations, but consider constraints among features unlike then, moreover it does not focus on faulty statements.

6.2.4 Configuration Troubleshooting

Some techniques have been recently proposed to deal with problematic configurations in configurable systems GARVIN; COHEN; DWYER (2013); ZHANG; ERNST (2013, 2014); SWANSON et al. (2014).

For instance, ConfDiagnoser ZHANG; ERNST (2013) and ConfSuggester ZHANG; ERNST (2014) propose a technique to troubleshoot configuration errors caused by configurable systems' evolution. They use dynamic profiling, execution trace comparison, and static analysis to link the undesired behavior to its root cause, a configuration option whose value can be changed to produce desired behavior from the new software version.

Garvin *et al.* GARVIN; COHEN; DWYER (2013) try to predict future failure-prone configurations in configurable system based on the history of this kind of failure; they use configuration similarity to make this prediction. Swanson *et al.* SWANSON et al. (2014)

builds on Garvin *et al.*'s approach, they propose an automated approach to detect failure and its workaround (the features that led to failure), and thus to reconfigure configurable systems in order to avoid future failures. These techniques complements `SPLif`, whose focus is on classifying tests and configurations for inspection.

7

Conclusion and Future Work

Software Product Lines (SPLs) are emerging as an important design and implementation principle for controlling variability in families of related software products. Testing SPLs is essential, and considerable recent research focuses on that topic. In the first part of this research, `SPLat` assumes that SPLs come equipped with complete, formally specified feature models. Unfortunately, this assumption does not always hold in practice. Thus, we proposed `SPLif` in order to solve the problems related to the lack (or incompleteness) of FMs.

Firstly, we proposed `SPLat`, a new technique for reducing the combinatorics in testing configurable systems. `SPLat` dynamically prunes the space of configurations that each test must be run against. `SPLat` minimizes the number of configurations and does so in a lightweight way compared to previous approaches based on static analysis and heavyweight dynamic execution. Experimental results on 10 software product lines written in Java show that `SPLat` substantially reduces the total test execution time in most cases. Moreover, our application of `SPLat` on a large industrial code written in Ruby on Rails shows its scalability.

Lastly, presented `SPLif`, a new approach for testing SPLs with incomplete feature models, or even no feature model at all. `SPLif` helps the user prioritize failing tests and configurations for inspection. Our experiments showed that `SPLif` is promising and can scale to large systems, such as GCC. In the near future, we plan to apply `SPLif` to other large configurable systems and to optimize execution by leveraging the similarities across multiple similar states D'AMORIM; LAUTERBURG; MARINOV (2008); KIM; KHURSHID; BATORY (2012); NGUYEN; KäSTNER; NGUYEN (2014).

7.1 Future Work

We have discussed many improvements in sections of threats to validity (3.3.6, 4.3.7 and 5.5) that deserve to be deeply investigated and implemented in future. In general, it is possible to improve the evaluation by applying `SPLif` to other large configurable system. It is also possible to optimize `SPLat` execution by leveraging the similarities across multiple similar states D'AMORIM; LAUTERBURG; MARINOV (2008); KIM; KHURSHID; BATORY (2012);

NGUYEN; KäSTNER; NGUYEN (2014). More specifically, there are two main research lines related to this work: feature model inference and regression testing of SPLs. The following we detail some possible works that we started to invest in these lines, but requires more research and development.

7.1.1 Feature Model Inference

Although substantial research has been done in inferring FM constraints, the problem is still under active investigation mainly due to the limitations of existing techniques. One way to complement this work would be inferring FM constraints from code by using static analysis. Another, and complementary, way would be inferring FM constraints from documentation, in order to face current limitations. This activity is fundamental to check the impact of the incompleteness of the FM in results.

7.1.1.1 Feature Model Inference from Code - SPLand

The main goal of this approach is to quickly identify tests that fail to consistent products. In this context, SPLand is similar to `SPLif`, but operates statically. The idea is that SPLand could provide support for `SPLif`, extracting constraints or relations between features, from source code, resulting in an initial model to serve as input to `SPLif`, instead of `SPLif` starting its execution using an empty model. Optimizing thus `SPLif` execution and reducing user effort both to validate products and to inspect code/test. And consequently, this process may result on a more complete feature model, resulting from SPLand and `SPLif`.

This approach intends to use static analysis techniques to extract relations between code members that are guarded by features, in order to infer constraints that will compose or complement the feature model. As an example, from the code in Figure 7.1, it is possible to extract the following constraint: $TOOLBAR \wedge WORDCOUNT \implies featureOf(wordCountButton)$, based on the control-flow, where *featureOf* indicates that feature *wordCountButton* belongs to features *TOOLBAR* and *WORDCOUNT*. Therefore, these features imply the feature that *wordCountButton* belongs to, that means, the variable *wordCountButton* can only be accessed if the features *TOOLBAR* and *WORDCOUNT* are enabled.

```
1 if(TOOLBAR){
2   if(WORDCOUNT){
3     wordCountButton.setToolTipText(Word count);
4   }
5 }
```

Figure 7.1: Peace of Notepad code.

7.1.1.2 Feature Model Inference from Documentation - MIHCO

The purpose of this approach is to uses the crowd to infer configuration constraints from *existing documentation*. As other crowd solving problems such as protein folding[1] and character recognition[2], this approach proposes to formulate problems/questions which are relatively simple for a large audience to solve but can be challenging for computers to solve. For that, MIHCO would require input from a diverse community of users to infer configuration constraints from short snippets of text original from existing documentation.

This proposal builds on the observation that long-standing popular configurable systems provide rich documentation on how to use configuration options for customization; documentation that has been scrutinized over time, certainly receiving feedback from a critical mass of users. Our assumption is that documentation becomes more complete and more precise as the system evolves with the active participation of a large community. For instance, the following popular configurable systems has been under active development for at least a decade: ApacheHTTP DOCUMENTATION (2014a), Firefox DOCUMENTATION (2014b), GCC DOCUMENTATION (2014c), and MySQL DOCUMENTATION (2014d).

The general idea is that MIHCO would work by extracting short snippets of text from existing documentation. Where, each snippet of text corresponds to a description of a feature option. If this description contains references for other options, we consider it as a question, that is given for a large audience which should infer the relationship between the options contained in the problem.

To illustrate how constraints can be inferred with this approach, we used GCC, which contains 227 feature options in the Optimization module, from which we could formulate 149 problems. Figure 7.2 illustrates a problem elaborated from the existing GCC documentation. This problem is given for crowd individuals that should infer a constraint like this: $\textit{-findirect-inlining} \Rightarrow (\textit{-finline-functions} \vee \textit{-finline-small-functions}) \wedge \textit{-O2}$.

From the following paragraph, write a propositional formula to characterize (possible) option relationships:
-findirect-inlining: Inline also indirect calls that are discovered to be known at compile time thanks to previous inlining. This option has any effect only when inlining itself is turned on by the **-finline-functions** or **-finline-small-functions** options. Enabled at level **-O2**.

Figure 7.2: An example of a problem extracted from the documentation of optimization module of GCC. This example can be find at: `http://www.cin.ufpe.br/~sfs/survey/first.php?question=1&user=10`.

It is important to note that documentation suffers from limitations (incompleteness and imprecision). Nevertheless, we highlight that: (1) documentation may contain constraints which

[1] http://folding.stanford.edu
[2] http://recaptcha.net

require specialized analysis for a particular language for detection or just can't be detected through static analysis at all NADI et al. (2014); (2) documentation is the amalgamating software artifact that has the potential to unite a diversity of people – end users, managers, and developers – around the key aspects of a configurable system. Learning from documentation can complement existing reverse-engineering techniques and also make the documentation itself better (once inferred models become useful).

7.1.2 Regression Testing of Software Product Lines

Automated tests are typically written to be used in regression testing, where they are run not only once but multiple times on evolving versions of the code under test. An important question is how to further speed up `SPLat` for regression testing. When we run a test for the first time, we can record which configurations were relevant for this test. We can compactly encode all these configurations as a decision tree for each test. Moreover, a number of tests can share these trees. Then, when the code under test (or the test code itself) is changed, we need not run the full `SPLat`() but can reuse the configuration tree from the previous run and execute only those configurations.

However, note that there is a challenge here that the change (in the code under test or the test itself) could have invalidated the set of relevant configurations. More precisely, there can be (1) configurations that are not encoded in the tree but became relevant after the change, and/or (2) configurations that are encoded in the tree but are not relevant any more after the change. One way to solve this problem would be to produce a new version of `SPLat` algorithm (RegressionSPLat) adapted for regression testing that can take an old tree, execute it on the new code, and do the following:

1. If the tree remained a precise encoding of all the relevant configurations, then RegressionSPLat finishes faster than `SPLat`;

2. If the tree is not precise any more, then RegressionSPLat can determine that change is required (*i.e.*, new configurations should be added and/or some existing configurations should be removed) and can build a new tree, while running only a bit slower than `SPLat`.

Since the case where the tree remains the same is much more common in the evolution of real code, the speedup gain from case 1 significantly outweighs the slowdown loss from case 2.

The intuition is that the algorithm RegressionSPLat could minimally instrument the accesses to feature variables in order to check that the variables accessed in the current exploration actually match what is in the tree from the previous exploration. If RegressionSPLat find any deviation, it can make all the `SPLat` exploration from scratch, or even better it could incrementally only run the `SPLat` exploration on the subtrees where RegressionSPLat found some change.

In general, the idea is to extend `SPLat` and `SPLif` to support the evolution of SPLs and configurable systems. The purpose is to speed up `SPLat` for regression testing, in order to avoid re-executing all tests at every system version or after changes. And `SPLif` can use information of regression test results to further avoid inspections. However, this last idea needs to be further investigated.

References

ABREU, R.; ZOETEWEIJ, P.; GEMUND, A. J. C. v. An Evaluation of Similarity Coefficients for Software Fault Localization. In: PACIFIC RIM INTERNATIONAL SYMPOSIUM ON DEPENDABLE COMPUTING, 12. **Proceedings...** [S.l.: s.n.], 2006. p.39–46. (PRDC '06).

ABREU, R.; ZOETEWEIJ, P.; GEMUND, A. J. C. v. Spectrum-Based Multiple Fault Localization. In: ASE '09. 24TH IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING. **Anais...** [S.l.: s.n.], 2009.

ABREU, R.; ZOETEWEIJ, P.; GEMUND, A. van. On the Accuracy of Spectrum-based Fault Localization. In: TESTING: ACADEMIC AND INDUSTRIAL CONFERENCE PRACTICE AND RESEARCH TECHNIQUES - MUTATION, 2007. TAICPART-MUTATION 2007. **Anais...** [S.l.: s.n.], 2007. p.89–98.

ACHER, M. et al. On Extracting Feature Models from Product Descriptions. In: SIXTH INTERNATIONAL WORKSHOP ON VARIABILITY MODELING OF SOFTWARE-INTENSIVE SYSTEMS. **Proceedings...** [S.l.: s.n.], 2012. p.45–54. (VaMoS '12).

AHO, A. V.; GRIFFETH, N. D. Feature Interactions in the Global Information Infrastructure. In: ACM SIGSOFT SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, 3. **Proceedings...** [S.l.: s.n.], 1995. p.2–4. (SIGSOFT '95).

AL-HAJJAJI, M. et al. Similarity-based Prioritization in Software Product-line Testing. In: SPLC. **Anais...** [S.l.: s.n.], 2014. p.197–206.

ALVES, V. et al. Extracting and Evolving Mobile Games Product Lines. In: INTERNATIONAL CONFERENCE ON SOFTWARE PRODUCT LINES, 9. **Proceedings...** [S.l.: s.n.], 2005. p.70–81. (SPLC'05).

ALVES, V. et al. An Exploratory Study of Information Retrieval Techniques in Domain Analysis. In: INTERNATIONAL SOFTWARE PRODUCT LINE CONFERENCE, 2008. SPLC '08., 12. **Anais...** [S.l.: s.n.], 2008. p.67–76.

ANDERSEN, N. et al. Efficient Synthesis of Feature Models. In: INTERNATIONAL SOFTWARE PRODUCT LINE CONFERENCE - VOLUME 1, 16. **Proceedings...** [S.l.: s.n.], 2012. p.106–115. (SPLC '12).

APEL, S.; BEYER, D. Feature Cohesion in Software Product Lines: An Exploratory Study. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 33. **Proceedings...** [S.l.: s.n.], 2011. p.421–430. (ICSE '11).

APEL, S. et al. Detection of Feature Interactions Using Feature-Aware Verification. In: AUTOMATED SOFTWARE ENGINEERING (ASE), 2011 26TH IEEE/ACM INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2011. p.372–375.

APEL, S. et al. Strategies for Product-Line Verification: case studies and experiments. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 2013. **Proceedings...** [S.l.: s.n.], 2013. p.482–491. (ICSE '13).

APEL, S. et al. **Feature-Oriented Software Product Lines: Concepts and Implementation**. Berlin/Heidelberg: [s.n.], 2013. 308 pages, ISBN 978-3-642-37520-0.

AUSTIN, T. H.; FLANAGAN, C. Multiple Facets for Dynamic Information Flow. In: ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 39., New York, NY, USA. **Proceedings...** ACM, 2012. p.165–178. (POPL '12).

BATORY, D. Feature Models, Grammars, and Propositional Formulas. In: INTERNATIONAL CONFERENCE ON SOFTWARE PRODUCT LINES, 9. **Proceedings...** [S.l.: s.n.], 2005. p.7–20. (SPLC'05).

BATORY, D.; SARVELA, J. N.; RAUSCHMAYER, A. Scaling Step-Wise Refinement. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 25., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2003. p.187–197. (ICSE '03).

BERGER, T. et al. Variability Modeling in the Real: A Perspective from the Operating Systems Domain. In: IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING. **Proceedings...** [S.l.: s.n.], 2010. p.73–82. (ASE '10).

BORBA, P. et al. Analysis, Test and Verification in the Presence of Variability (Dagstuhl Seminar 13091). **Dagstuhl Reports**, Dagstuhl, Germany, v.3, n.2, p.144–170, 2013.

BOYAPATI, C.; KHURSHID, S.; MARINOV, D. Korat: Automated Testing Based on Java Predicates. In: ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS, 2002. **Proceedings...** [S.l.: s.n.], 2002. p.123–133. (ISSTA '02).

CABRAL, I.; COHEN, M. B.; ROTHERMEL, G. Improving the Testing and Testability of Software Product Lines. In: INTERNATIONAL CONFERENCE ON SOFTWARE PRODUCT LINES: GOING BEYOND, 14. **Proceedings...** [S.l.: s.n.], 2010. p.241–255. (SPLC'10).

CAMPOS, J. C. de et al. Entropy-Based Test Generation for Improved Fault Localization. In: AUTOMATED SOFTWARE ENGINEERING (ASE), 2013 IEEE/ACM 28TH INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2013. p.257–267.

CHANDRA, S. et al. Angelic Debugging. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 33. **Proceedings...** [S.l.: s.n.], 2011. p.121–130. (ICSE '11).

CHEN, Y. et al. Taming Compiler Fuzzers. In: ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, 34. **Proceedings...** [S.l.: s.n.], 2013. p.197–208. (PLDI '13).

CLASSEN, A. et al. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In: ND ACM/IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING - VOLUME 1, 32. **Proceedings...** [S.l.: s.n.], 2010. p.335–344. (ICSE '10).

CLEMENTS, P.; NORTHROP, L. M. **Software Product Lines**: practices and patterns. [S.l.]: Addison-Wesley, 2001. (Professional).

COHEN, M. B.; DWYER, M. B.; SHI, J. Coverage and Adequacy in Software Product Line Testing. In: ISSTA 2006 WORKSHOP ON ROLE OF SOFTWARE ARCHITECTURE FOR TESTING AND ANALYSIS. **Proceedings...** [S.l.: s.n.], 2006. p.53–63. (ROSATEA '06).

COHEN, M. B.; DWYER, M. B.; SHI, J. Interaction testing of highly-configurable systems in the presence of constraints. In: ISSTA '07: PROCEEDINGS OF THE 2007 INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS, New York, NY, USA. **Anais...** ACM, 2007. p.129–139.

CZARNECKI, K.; EISENECKER, U. **Generative programming**: methods, tools, and applications. [S.l.]: Addison-Wesley, 2000.

CZARNECKI, K.; HELSEN, S.; EISENECKER, U. W. Formalizing Cardinality-Based Feature Models and Their Specialization. **Software Process: Improvement and Practice**, [S.l.], v.10, n.1, p.7–29, 2005.

CZARNECKI, K.; WASOWSKI, A. Feature Diagrams and Logics: There and Back Again. In: INTERNATIONAL SOFTWARE PRODUCT LINE CONFERENCE, 11. **Proceedings...** [S.l.: s.n.], 2007. p.23–34. (SPLC '07).

DALLMEIER, V.; LINDIG, C.; ZELLER, A. Lightweight Defect Localization for Java. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 19. **Proceedings...** [S.l.: s.n.], 2005. p.528–550. (ECOOP'05).

D'AMORIM, M.; LAUTERBURG, S.; MARINOV, D. Delta Execution for Efficient State-space Exploration of Object-oriented Programs. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS, 2007. **Proceedings...** [S.l.: s.n.], 2007. p.50–60. (ISSTA '07).

D'AMORIM, M.; LAUTERBURG, S.; MARINOV, D. Delta Execution for Efficient State-Space Exploration of Object-Oriented Programs. **IEEE TSE**, [S.l.], v.34, n.5, p.597–613, 2008.

DANIEL, B.; GVERO, T.; MARINOV, D. On Test Repair Using Symbolic Execution. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS, 19. **Proceedings...** [S.l.: s.n.], 2010. p.207–218. (ISSTA '10).

DAVRIL, J.-M. et al. Feature Model Extraction from Large Collections of Informal Product Descriptions. In: JOINT MEETING ON FOUNDATIONS OF SOFTWARE ENGINEERING, 2013. **Proceedings...** [S.l.: s.n.], 2013. p.290–300. (ESEC/FSE 2013).

DejaGnu. **DejaGnu**.
`http://www.gnu.org/software/dejagnu/`.

DejaGnu directives. **Syntax and Descriptions of test directives**.
`http:`
`//gcc.gnu.org/onlinedocs/gccint/Directives.html#Directives`.

DOCUMENTATION, F.
`http://preferential.mozdev.org/preferences.html`.

DOCUMENTATION, G.
`https://gcc.gnu.org/onlinedocs/gcc/Option-Index.html`.

DOCUMENTATION, h.
`http://httpd.apache.org/docs/current/mod/directives.html`.

DOCUMENTATION, M.
http://dev.mysql.com/doc/refman/5.0/en/dynindex-option.html.

FEST. **FEST: fixtures for easy software testing**.
http://fest.easytesting.org/.

GARVIN, B.; COHEN, M. Feature Interaction Faults Revisited: An Exploratory Study. In: IEEE 22ND INTERNATIONAL SYMPOSIUM ON SOFTWARE RELIABILITY ENGINEERING (ISSRE). **Anais...** [S.l.: s.n.], 2011. p.90–99.

GARVIN, B.; COHEN, M.; DWYER, M. Failure Avoidance in Configurable Systems through Feature Locality. In: **Assurances for Self-Adaptive Systems**. [S.l.: s.n.], 2013. p.266–296. (Lecture Notes in Computer Science, v.7740).

GCC - Test. **Preparing Testcases**.
http://gcc.gnu.org/wiki/HowToPrepareATestcase.

GCC. **GCC, the GNU Compiler Collection**.
http://gcc.gnu.org/.

GCC Documentation. **Options That Control Optimization**.
gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#
Optimize-Options.

GCC Options. **Option Summary**.
http://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html.

GHANDEHARI, L. et al. Fault Localization Based on Failure-Inducing Combinations. In: IEEE 24TH INTERNATIONAL SYMPOSIUM ON SOFTWARE RELIABILITY ENGINEERING (ISSRE). **Anais...** [S.l.: s.n.], 2013. p.168–177.

GHEYI, R.; MASSONI, T.; BORBA, P. Algebraic Laws for Feature Models. **Journal of Universal Computer Science**, [S.l.], v.14, n.21, p.3573–3591, 2008.

GODEFROID, P. Model Checking for Programming Languages Using VeriSoft. In: ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 24. **Proceedings...** [S.l.: s.n.], 1997. p.174–186. (POPL '97).

GUIDSL.
http://www.cs.utexas.edu/~schwartz/ATS/fopdocs/guidsl.html.

HALL, R. Fundamental Nonmodularity in Electronic Mail. **Automated Software Engineering**, [S.l.], v.12, n.1, p.41–79, 2005.

HASLINGER, E. N.; LOPEZ-HERREJON, R. E.; EGYED, A. On Extracting Feature Models from Sets of Valid Feature Combinations. In: INTERNATIONAL CONFERENCE ON FUNDAMENTAL APPROACHES TO SOFTWARE ENGINEERING (FASE). **Proceedings...** [S.l.: s.n.], 2013. p.53–67.

HENARD, C. et al. Towards Automated Testing and Fixing of Re-engineered Feature Models. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 2013. **Proceedings...** [S.l.: s.n.], 2013. p.1245–1248. (ICSE '13).

HOSEK, P.; CADAR, C. Safe Software Updates via Multi-version Execution. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE 2013). **Anais...** [S.l.: s.n.], 2013.

Human-resource management system. **101Companies**. `http://101companies.org/index.php/101companies:Project`.

Instructions to students on generating new tests for SPLs. `http://www.cin.ufpe.br/~sfs/splif/experiments.html`.

Java tokenizer and parser tools. **JTopas**. `http://jtopas.sourceforge.net/jtopas/index.html`.

JONES, J. A.; HARROLD, M. J.; STASKO, J. Visualization of test information to assist fault localization. In: ICSE. **Anais...** [S.l.: s.n.], 2002.

KANG, K. et al. **Feature-Oriented Domain Analysis (FODA) Feasibility Study**. [S.l.]: Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990. Technical Report. (CMU/SEI-90-TR-21).

KASTNER., C. **Virtual Separation of Concerns**: toward preprocessors 2.0. 2010. Tese (Doutorado em Ciência da Computação) — Otto-von-Guericke-Universitat Magdeburg.

KäSTNER, C. et al. Toward Variability-Aware Testing. In: INTERNATIONAL WORKSHOP ON FEATURE-ORIENTED SOFTWARE DEVELOPMENT, 4. **Proceedings...** [S.l.: s.n.], 2012. p.1–8. (FOSD '12).

KICZALES, G. et al. An Overview of AspectJ. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 15. **Proceedings...** [S.l.: s.n.], 2001. p.327–353. (ECOOP '01).

KIM, C. H. P.; BATORY, D. S.; KHURSHID, S. Reducing C in Testing Product Lines. In: TENTH INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT. **Proceedings...** [S.l.: s.n.], 2011. p.57–68. (AOSD '11).

KIM, C. H. P. et al. Reducing Configurations to Monitor in a Software Product Line. In: RV. **Anais...** [S.l.: s.n.], 2010. p.285–299.

KIM, C. H. P. et al. SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In: FOUNDATIONS OF SOFTWARE ENGINEERING, 2013. **Proceedings...** [S.l.: s.n.], 2013. p.257–267. (ESEC/FSE 2013).

KIM, C.; KHURSHID, S.; BATORY, D. Shared Execution for Efficiently Testing Product Lines. In: SOFTWARE RELIABILITY ENGINEERING (ISSRE), 2012 IEEE 23RD INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2012. p.221–230.

KOLBITSCH, C. et al. Rozzle: De-cloaking Internet Malware. In: IEEE SYMPOSIUM ON SECURITY AND PRIVACY, 2012. **Proceedings...** [S.l.: s.n.], 2012. p.443–457. (SP '12).

KORAT Home Page. `http://mir.cs.illinois.edu/korat/`.

KRAMER, J. et al. CONIC: an integrated approach to distributed computer control systems. **Computers and Digital Techniques, IEE Proceedings E**, [S.l.], v.130, n.1, January 1983.

LE, V.; AFSHARI, M.; SU, Z. Compiler Validation via Equivalence Modulo Inputs. In: ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, 35. **Proceedings...** [S.l.: s.n.], 2014. p.216–226. (PLDI '14).

LENGAUER, P. et al. Where Has All My Memory Gone?: Determining Memory Characteristics of Product Variants Using Virtual-machine-level Monitoring. In: EIGHTH INTERNATIONAL WORKSHOP ON VARIABILITY MODELLING OF SOFTWARE-INTENSIVE SYSTEMS. **Proceedings...** [S.l.: s.n.], 2013. p.1–8. (VaMoS).

Library for object persistence. **Prevayler**. [S.l.: s.n.], 2013. `http://prevayler.org/`.

Library to serialize objects to XML and back again. **XStream**. `http://xstream.codehaus.org/`.

LOPEZ-HERREJON, R. E.; BATORY, D. S. A Standard Problem for Evaluating Product-Line Methodologies. In: THIRD INTERNATIONAL CONFERENCE ON GENERATIVE AND COMPONENT-BASED SOFTWARE ENGINEERING. **Proceedings...** Springer, 2001. p.10–24. (GCSE '01).

LOPEZ-HERREJON, R. E. et al. Reverse Engineering Feature Models with Evolutionary Algorithms: An Exploratory Study. In: INTERNATIONAL CONFERENCE ON SEARCH BASED SOFTWARE ENGINEERING, 4. **Proceedings...** [S.l.: s.n.], 2012. p.168–182. (SSBSE'12).

MCGREGOR, J. **Testing a Software Product Line**. [S.l.]: CMU/SEI, 2001. Available from `http://www.sei.cmu.edu/pub/documents/01.reports/pdf/01tr022.pdf`. (CMU/SEI-2001-TR-022).

NADI, S. et al. Mining Configuration Constraints: static analyses and empirical results. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 36. **Proceedings...** [S.l.: s.n.], 2014. p.140–151. (ICSE 2014).

NGUYEN, H. V.; KäSTNER, C.; NGUYEN, T. N. Exploring Variability-aware Execution for Testing Plugin-based Web Applications. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 36. **Proceedings...** [S.l.: s.n.], 2014. p.907–918. (ICSE 2014).

PLATH, M.; RYAN, M. Feature Integration Using a Feature Construct. **Sci. Comput. Program.**, [S.l.], v.41, n.1, p.53–84, 2001.

POHL, K.; BÖCKLE, G.; LINDEN, F. van der. **Software Product Line Engineering**: foundations, principles and techniques. [S.l.]: Springer, 2005.

POST, H.; SINZ, C. Configuration Lifting: Verification Meets Software Configuration. In: IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, 2008. **Proceedings...** [S.l.: s.n.], 2008. p.347–350. (ASE '08).

QU, X.; COHEN, M. B.; ROTHERMEL, G. Configuration-aware Regression Testing: An Empirical Study of Sampling and Prioritization. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS, 2008. **Proceedings...** [S.l.: s.n.], 2008. p.75–86. (ISSTA '08).

RABKIN, A.; KATZ, R. Static Extraction of Program Configuration Options. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 33. **Proceedings...** [S.l.: s.n.], 2011. p.131–140. (ICSE '11).

REISNER, E. et al. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In: ND ACM/IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING - VOLUME 1, 32. **Proceedings...** [S.l.: s.n.], 2010. p.445–454. (ICSE '10).

RENIERIS, M.; REISS, S. P. Fault Localization With Nearest Neighbor Queries. In: INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE). **Anais...** [S.l.: s.n.], 2003. p.30–39.

RHEIN, A. V.; APEL, S.; RAIMONDI, F. Introducing Binary Decision Diagrams in the Explicit-State Verification of Java Code. In: JPF WORKSHOP. **Anais...** [S.l.: s.n.], 2011. Available from `http://babelfish.arc.nasa.gov/trac/jpf/attachment/wiki/events/events/workshop2011/RheinBdd_sub.pdf`.

SAT4J. `http://www.sat4j.org/`.

SCHMID, K.; RABISER, R.; GRüNBACHER, P. A Comparison of Decision Modeling Approaches in Product Lines. In: WORKSHOP ON VARIABILITY MODELING OF SOFTWARE-INTENSIVE SYSTEMS, 5. **Proceedings...** [S.l.: s.n.], 2011. p.119–126. (VaMoS '11).

SCHOBBENS, P.-Y. et al. Generic Semantics of Feature Diagrams. **Comput. Netw.**, [S.l.], v.51, n.2, p.456–479, 2007.

SCHULER, D.; DALLMEIER, V.; ZELLER, A. Efficient Mutation Testing by Checking Invariant Violations. In: EIGHTEENTH INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS. **Proceedings...** [S.l.: s.n.], 2009. p.69–80. (ISSTA '09).

SEGURA, S. et al. Automated Test Data Generation on the Analyses of Feature Models: A Metamorphic Testing Approach. In: THIRD INTERNATIONAL CONFERENCE ON SOFTWARE TESTING, VERIFICATION AND VALIDATION (ICST). **Anais...** [S.l.: s.n.], 2010. p.35–44.

SHE, S. et al. Reverse Engineering Feature Models. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 33. **Proceedings...** [S.l.: s.n.], 2011. p.461–470. (ICSE '11).

SHI, J.; COHEN, M. B.; DWYER, M. B. Integration Testing of Software Product Lines Using Compositional Symbolic Execution. In: INTERNATIONAL CONFERENCE ON FUNDAMENTAL APPROACHES TO SOFTWARE ENGINEERING (FASE). **Proceedings...** [S.l.: s.n.], 2012. p.270–284.

SONG, C.; PORTER, A.; FOSTER, J. S. iTree: Efficiently Discovering High-Coverage Configurations Using Interaction Trees. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 34. **Proceedings...** [S.l.: s.n.], 2012. p.903–913. (ICSE '12).

SOUTO, S. et al. Faster Bug Detection for Software Product Lines with Incomplete Feature Models. In: INTERNATIONAL CONFERENCE ON SOFTWARE PRODUCT LINES, 19. **Proceedings...** [S.l.: s.n.], 2015. p.to–appear. (SPLC'15).

SWANSON, J. et al. Beyond the Rainbow: Self-adaptive Failure Avoidance in Configurable Systems. In: ND ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, 22. **Proceedings...** [S.l.: s.n.], 2014. p.377–388. (FSE 2014).

THAKER, S. et al. Safe Composition of Product Lines. In: INTERNATIONAL CONFERENCE ON GENERATIVE PROGRAMMING AND COMPONENT ENGINEERING, 6. **Proceedings...** [S.l.: s.n.], 2007. p.95–104. (GPCE '07).

TILLMANN, N.; SCHULTE, W. **Unit Tests Reloaded**: Parameterized Unit Testing with Symbolic Execution. Redmond, Washington: Microsoft Research, 2005. Technical Report. (MSR-TR-2005-153).

TUCEK, J.; XIONG, W.; ZHOU, Y. Efficient Online Validation with Delta Execution. In: ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, 14. **Proceedings...** [S.l.: s.n.], 2009. p.193–204. (ASPLOS '09).

UZUNCAOVA., E. **Efficient Specification-based Testing Using Incremental Techniques**. 2008. Tese (Doutorado em Ciência da Computação) — Department of Electrical and Computer Engineering, University of Texas at Austin.

WEBSITE, S. P. L. **Puzzle game**. [S.l.: s.n.], 2013. `https://code.launchpad.net/~spl-devel/spl/default-branch`.

WEISS, D. M. et al. Decision-Model-Based Code Generation for SPLE. In: INTERNATIONAL SOFTWARE PRODUCT LINE CONFERENCE, 2008. SPLC '08., 12. **Anais...** [S.l.: s.n.], 2008. p.129–138.

WESTON, N.; CHITCHYAN, R.; RASHID, A. A Framework for Constructing Semantically Composable Feature Models from Natural Language Requirements. In: INTERNATIONAL SOFTWARE PRODUCT LINE CONFERENCE, 13. **Proceedings...** [S.l.: s.n.], 2009. p.211–220. (SPLC '09).

XU, T. et al. Do Not Blame Users for Misconfigurations. In: TWENTY-FOURTH ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES. **Proceedings...** [S.l.: s.n.], 2013. p.244–259. (SOSP '13).

YANG, X. et al. Finding and Understanding Bugs in C Compilers. In: ND ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, 32. **Proceedings...** [S.l.: s.n.], 2011. p.283–294. (PLDI '11).

YOUNG, T. **Using AspectJ to Build a Software Product Line for Mobile Devices**. 2005. Dissertação (Mestrado em Ciência da Computação) — University of British Columbia, British Columbia, Canada.

Z3 Theorem Prover. `http://research.microsoft.com/en-us/um/redmond/projects/z3/`.

ZAVE, P. Feature Interactions and Formal Specifications in Telecommunications. **Computer**, [S.l.], v.26, n.8, p.20–28, Aug 1993.

ZELLER, A. Yesterday, My Program Worked. Today, It Does Not. Why? In: EUROPEAN SOFTWARE ENGINEERING CONFERENCE HELD JOINTLY WITH THE 7TH ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, 7. **Proceedings...** [S.l.: s.n.], 1999. p.253–267. (ESEC/FSE-7).

ZHANG, S.; ERNST, M. D. Automated Diagnosis of Software Configuration Errors. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 2013. **Proceedings...** [S.l.: s.n.], 2013. p.312–321. (ICSE '13).

ZHANG, S.; ERNST, M. D. Which Configuration Option Should I Change? In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 36. **Proceedings...** [S.l.: s.n.], 2014. p.152–163. (ICSE 2014).