PhD-FSTC-2015-17

The Faculty of Science, Technology and Communication

DISSERTATION

Defense held on the 27th May 2015 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

Christopher HENARD

Born on 8th June 1986 in Nancy (Lorraine, France)

ENABLING TESTING OF LARGE SCALE HIGHLY CONFIGURABLE SYSTEMS WITH SEARCH-BASED SOFTWARE ENGINEERING: THE CASE OF MODEL-BASED SOFTWARE PRODUCT LINES

Dissertation Defense Committee

Dr-Ing. Yves LE TRAON, Dissertation Supervisor
Professor, University of Luxembourg, Luxembourg

Dr. Lionel C. BRIAND, Chairman
Professor, University of Luxembourg, Luxembourg

Dr. Michail PAPADAKIS, Vice Chairman
Research Associate, University of Luxembourg, Luxembourg

Dr. Myra B. COHEN
Associate Professor, University of Nebraska, Lincoln, NE, USA

Dr-Ing. Jean-Marc JÉZÉQUEL
Professor, Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA), Rennes, France

Enabling testing of large scale highly configurable systems with search-based software engineering: the case of model-based software product lines

Abstract
Complex situations formed by mixes of versatile environments, various user needs and time-to-market constraints led to the development of highly configurable systems. In line with the emergence of such systems, software development is increasingly moving from the production of a single, yet configurable software to the development of families of software products. Such families of related software are called Software Product Lines (SPLs), and they allow the automation of the configuration, deployment and management of tailored software products through the combination of software features. These features and the constraints defining their legal combinations are usually encoded in a feature model (FM), which is used to represent a SPL.

One main challenge with SPLs is testing them, a task which is even more difficult as the number of features proposed is important. Ideally, all the possible products that can be configured from a SPL should be tested. This, however, is unfeasible in practice since only 270 optional features allows configuring more products than the number of atoms in the universe. Considering that realistic SPLs involve thousands of features and that testing capabilities are limited by time and budget constraints, only a subset of all the configurable products can actually be tested, introducing the needs for strategies to test such SPLs.

To reduce the testing effort, techniques using combinatorial interaction testing (CIT) have been proposed and proven to be successful. However, they fail at scaling to large and heavily constrained SPLs. In addition, CIT is costly to apply due to the combinatorial explosion induced by calculating the feature combinations. Besides, existing approaches do not consider multiple and potentially conflicting testing objectives such as minimizing the number of configurations and their cost. In this respect, this dissertation introduces scalable techniques for both generating and prioritizing relevant SPL product configurations for CIT by using a similarity heuristic which avoids the combinatorial explosion. In a second step, methods for handling multiple testing objectives are presented.

The following part of this thesis focuses on the quality assessment of given product configurations prior to testing. The objective here is to evaluate how good is a given set of configurations according to different testing criteria, whatever the way these configurations have been selected. This situation arises when configurations that have to be tested are already available. Since testing these software products individually is a costly and time consuming task, methodologies to evaluate them prior testing are introduced, thus allowing to discard unnecessary ones and save testing sessions. In particular, an approach based on mutation of the SPL FM which can form viable and cheaper alternative to CIT is presented.

The next part of this dissertation investigates the reverse-engineering of a SPL and its FM from existing source code of software product variants. Since SPLs allows us to reduce development costs and quickly derive tailored products for specific market share, automated techniques to migrate similar product variants into a whole SPL are necessary. In particular, the challenge of reverse-engineering a SPL which is concordant with the underlying software products is tackled by a fully automated approach. In addition, since reverse-engineering approaches (whether manually or automatically performed) are inherently error-prone, a methodology for evaluating and fixing reverse-engineered SPL FM is presented.

The final part of this dissertation describes the application of the introduced theoretical advances to an industrial case with the CETREL company. In this project, a credit card authorization system is tested by using credit card authorizations. The testing process is optimized by modeling credit card authorizations as a SPL, enabling the application of the above-mentioned generation and evaluation approaches. All the proposed approaches use search-based techniques combined with constraint solvers and have been validated through rigorous experiments performed on moderate to large scale SPLs.

Acknowledgements

This PhD is the result of a long process that began mid-October 2011 and whose outcome owes much to the support and help of a few people. It is a pleasure for me to express my gratitude to them.

First of all, I am grateful to my supervisor, Yves Le Traon, for allowing me to perform this thesis after my graduation internship. He always trusted and supported me throughout these three and a half years. I would like particularly to thank him for having reassured and encouraged me to continue in the moments of doubts that I had, and for always emphasizing the bright side of things.

I am equally thankful to my daily supervisor, Mike Papadakis, for his patience, advice and flawless guidance. He pointed me in the right direction, showed me how to perform research, taught me how to write papers and how to conduct rigorous experiments. I am very happy about the friendship we have build up during the years.

Of all the people involved in my thesis, I am particularly grateful to Gilles Perrouin and Jacques Klein. They somehow initiated my work on the software product line subject and introduced me to all the related concepts. I had the chance to work with them and I really appreciate that they were always here to support me when I was down. I also would like to thank all the jury members for their interest in my research and for the time they invested in my dissertation.

I am thankful to Renaud Dechambre and Alain Barthelemy from CETREL for the time they found to collaborate with us. It was really useful and rewarding to me to be able to apply my research to a concrete industrial case.

I would like to express my warm thanks to the all group members of SERVAL for the plenty good coffee breaks and discussions we have had. I would like to extend my thanks to all my co-authors and to the people I have been in touch during my PhD and that I'm not explicitly citing.

Finally, and more personally, I would like to express my deepest heartfelt thanks to my family and my friends for their support, especially to my girlfriend Birsena for her love and support and to my mother who led a life of sacrifice to allow me to study in the best possible conditions.

Christopher Hénard

Luxembourg, Luxembourg, April 2015

Contents

List of abbreviations & acronyms xi

List of figures xiii

List of tables xv

List of algorithms xvii

1 General introduction 1
1.1 Context . . . 2
1.1.1 Highly configurable systems and software product lines . . . 2
1.1.2 Terminology . . . 4
1.1.3 Increasing adoption of software product lines . . . 4
1.2 Issues and contributions . . . 4
1.2.1 Challenges . . . 5
1.2.2 Overview of the contributions and organization of the dissertation . . . 6

I Background and state of the art 9

2 Technical background and definitions 11
2.1 Model-based software product lines . . . 12
2.1.1 Software product line engineering . . . 12
2.1.2 Feature models . . . 12
2.1.3 Configurations . . . 13
2.1.4 Mutation analysis . . . 15
2.2 Combinatorial interaction testing . . . 15
2.2.1 T-wise testing and coverage . . . 16
2.2.2 The impact of constraints . . . 16
2.2.3 Combinatorial interaction testing models . . . 17
2.3 Optimization problems . . . 17
2.3.1 Metaheuristics . . . 17
2.3.2 Search-based software engineering . . . 17
2.3.3 Multi-objective optimization . . . 19
2.3.4 Configuration generation and prioritization . . . 19

3 Related work 23
3.1 Configuration generation and prioritization . . . 24
3.1.1 T-wise generation . . . 24
3.1.2 Multi-objective approaches . . . 26
3.1.3 Generating configurations from a feature model with constraint solvers . . . 27
3.2 Configuration evaluation and mutation . . . 28
3.2.1 Mutation analysis . . . 28
3.2.2 Fault detection ability . . . 28
3.3 Reverse-engineering and re-engineering software product lines . . . 29
3.3.1 Extracting a software product line . . . 29
3.3.2 Automated improvement and fixing of extracted software product lines . . . 30

II Mono-objective configuration generation 31

4 Scalable t-wise generation and prioritization of software product line configurations 33
4.1 Introduction . . . 34
4.2 The similarity heuristic . . . 35
4.3 Configuration generation . . . 36
4.3.1 A similarity-based fitness function . . . 36
4.3.2 A search-based approach . . . 37
4.4 Configuration prioritization . . . 38
4.4.1 Local Maximum Distance prioritization . . . 38
4.4.2 Global Maximum Distance prioritization . . . 39
4.5 Empirical study . . . 39
4.5.1 Comparison with state of the art tools (research question 1) . . . 41
4.5.2 Configuration generation assessment (research question 2) . . . 43
4.5.3 Configuration prioritization assessment (research questions 3 & 4) . . . 48
4.6 Discussion . . . 52
4.6.1 Detecting t-wise interaction faults . . . 52
4.6.2 Practical implications . . . 54
4.6.3 Further applications . . . 55
4.6.4 Limitations . . . 55
4.6.5 Threats to validity . . . 56
4.7 Conclusions . . . 56

5 A mutation-based approach for generating software product line configurations 57
5.1 Introduction . . . 58
5.2 The mutation-based configuration generation approach . . . 58
5.2.1 Creation of mutants of the feature model . . . 59
5.2.2 The search-based process . . . 59
5.3 Experiments . . . 61
5.3.1 Approach assessment (research question 1) . . . 62
5.3.2 Comparison with random (research question 2) . . . 64
5.4 Threats to validity . . . 66
5.5 Conclusions . . . 67

III Multi-objective configuration generation 69

6 A constraint-aware search approach for configuring large software product lines 71
6.1 Introduction . . . 72
6.2 The existing approaches . . . 73
6.3 The proposed approaches . . . 73
6.3.1 Diversity promotion . . . 74
6.3.2 “Smart" operators . . . 74
6.3.3 The SATIBEA approach . . . 75
6.3.4 The Filtered approach . . . 75
6.4 Research questions . . . 76
6.5 Experimental setup . . . 77
6.5.1 Subjects . . . 77
6.5.2 Optimization objectives . . . 77
6.5.3 Settings . . . 78
6.5.4 Metrics . . . 78
6.5.5 Statistical analysis and tests . . . 81

6.6 Experimental results . . . 81
6.6.1 Results . . . 81
6.6.2 Answering research questions 1 & 2 . . . 82
6.6.3 Answerwing research question 3 . . . 82
6.6.4 Answering research question 4 . . . 83
6.7 Discussion . . . 83
6.7.1 Practical implications . . . 83
6.7.2 Threats to validity . . . 84
6.8 Conclusions . . . 85

7 Handling multiple testing objectives targeting a whole configuration suite 87
7.1 Introduction . . . 88
7.2 The multi-objective configuration generation approach . . . 88
7.2.1 Modeling individuals and population . . . 88
7.2.2 Modeling the genetic algorithm operations . . . 89
7.2.3 The objective function . . . 89
7.2.4 Overview of the approach . . . 90
7.3 Case study . . . 92
7.3.1 Approach parameters . . . 92
7.3.2 Evaluation of the objective function **F** (research question 1) . . . 93
7.3.3 Comparison with Random (research question 2) . . . 95
7.4 Threats to validity . . . 97
7.5 Conclusions . . . 98

IV Configuration evaluation 99

8 Assessing configurations with mutation and application to similarity testing 101
8.1 Introduction . . . 102
8.2 Approach . . . 103
8.2.1 The mutation analysis approach for evaluating configurations . . . 103
8.2.2 configuration suite generation . . . 103
8.2.3 Evaluation of the quality of the configuration suite . . . 104
8.3 Experiments . . . 104
8.3.1 Evaluation of the mutation score depending on the type of configurations . . . 104
8.3.2 Impact of the similarity-based prioritization on the mutation score . . . 106
8.3.3 Answering research questions 1 and 2 . . . 108
8.3.4 Threats to validity . . . 109
8.4 Conclusions . . . 109

9 Mutation analysis as a potential alternative to combinatorial interaction testing 111
9.1 Introduction . . . 112
9.2 Example . . . 113
9.2.1 Case 1: absence of input constraints . . . 113
9.2.2 Case 2: presence of input constraints . . . 115
9.3 Test suite evaluation . . . 115
9.3.1 The program input model . . . 115
9.3.2 The combinatorial interaction testing approach . . . 116
9.3.3 The mutation approach . . . 116
9.4 Experimental methodology . . . 117
9.4.1 Definition of the experiment . . . 117
9.4.2 Subjects . . . 118
9.4.3 Evaluating test suites . . . 118

9.4.4 Rank correlation analysis 119
9.5 Experimental results 120
9.5.1 Correlation analysis (research questions 1 & 2) 120
9.5.2 Comparing combinatorial testing and mutation (research question 3) 120
9.6 Discussion 122
9.6.1 Additional consideration about mutation and combinatorial testing 123
9.6.2 Cost of the approach 123
9.6.3 Threats to validity 124
9.7 Conclusions 124

V Reverse-engineering and re-engineering 125

10 From software product variants to a software product line: a preliminary approach 127
10.1 Introduction 128
10.2 The ExtractorPL approach 128
10.2.1 The banking system example 129
10.2.2 Abstraction of the software product variants 129
10.2.3 Automatic identification of the software product line features 131
10.2.4 Feature code generation 133
10.3 Case study 133
10.3.1 Accuracy of the extracted software product line (research question 1) 134
10.3.2 Quality of the extracted software product line (research question 2) 135
10.3.3 Threats to validity 137
10.4 Discussion 137
10.4.1 Benefits 137
10.4.2 Limitations 138
10.5 Conclusions 138

11 Fixing re-engineered software product line feature models 141
11.1 Introduction 142
11.2 Test-and-fix loop 142
11.2.1 Testing the feature model 143
11.2.2 Fixing the feature model 144
11.2.3 Continuous improvement 144
11.3 Preliminary evaluation 145
11.3.1 The Linux kernel feature model 145
11.3.2 Evaluation of the re-engineered Linux kernel feature model 145
11.3.3 Improving the feature model 146
11.4 Conclusions 147

VI Industrial application and final remarks 149

12 Testing card authorization systems with CETREL: a real-world case study 151
12.1 Context 152
12.1.1 Migrating to a new card authorization systems 152
12.1.2 A difficult migration 152
12.1.3 Improving the testing process 153
12.2 Approach 153
12.2.1 Modeling authorizations as a product line 154
12.2.2 Application of configuration generation and evaluation approaches 154

12.3 Preliminary results and future investigations . . . 155
12.3.1 Current results . . . 156
12.3.2 Work in progress and perspectives . . . 156
12.4 Conclusions . . . 157

13 Conclusions and outlook **159**
13.1 Summary . . . 160
13.2 Future work and open research questions . . . 160
13.2.1 Search-based techniques applied to software product lines . . . 160
13.2.2 Combinatorial interaction testing in practice . . . 161

List of papers, tools & services **163**

Bibliography **165**

List of abbreviations & acronyms

CIT	Combinatorial interaction testing
CNF	Conjunctive normal form
CP	Construction primitive
CS	Configuration suite
FM	Feature model
FST	Feature structure tree
HCS	Highly configurable system
HV	Hypervolume
IBEA	Indicator-based evolutionary algorithm
IGD	Inverted generational distance
MOO	Multi-objective optimization
MS	Mutation-score
N/A	Not available
PFS	Pareto front size
RQ	Research question
S	Spread
SAT	Satisfiability
SB	Search-based
SBSE	Search-based software engineering
SMT	Satisfiability modulo theory
SoCPs	Set of construction primitives
SP	Software product
SPL	Software product line
SPLE	Software product line engineering
SPVs	Software product variants

List of figures

1 General introduction **2**
1.1 Configuration interface for the Linux kernel v2.2.6 2
1.2 Example of (software) product line. 3
1.3 Simplified view of the configuration process in configurable systems and software product lines 3
1.4 Issues addressed in the dissertation. 5
1.5 Structure of the dissertation. 7

2 Technical background and definitions **12**
2.1 A feature model of a raster graphics editor software product lines 12
2.2 The process of evolving a population in genetic algorithms 18
2.3 An example of Pareto front with two objectives F_1 and F_2 to minimize. 19
2.4 A different ordering of the configurations allows reaching faster or slower the t-wise coverage provided by these configurations. 20

4 Scalable t-wise generation and prioritization of software product line configurations **34**
4.1 Configuration generation on the 110 moderate feature models for $t = 2$. 44
4.2 Result of the Mann-Whitney U Test between the search-based and the unpredictable approach for $t = 2...6$ on the four large feature models (equal hypothesis). 47
4.3 Fitness function correlation with t-wise coverage for the Linux kernel feature model. 47
4.4 Prioritization on the moderate size feature models for $t = 2$. 51
4.5 Execution time for the prioritization on the moderate size feature models for $t = 2$. . 52
4.6 Global Maximum Distance VS Random prioritization on the four large feature models. 53
4.7 Estimation of the interaction fault detection rate of the Global Maximum Distance and Random prioritizations for all the t-values. 53

5 A mutation-based approach for generating software product line configurations **58**
5.1 Overview of the mutation-based approach for generating configurations. 59
5.2 The search operators used by the generation approach. 61
5.3 Evolution of the mutation score over the 1,000 generations averaged on all the feature models for the 30 runs. 63
5.4 Search-based approach VS Random: distribution of the mutation score and number of configurations on the 30 runs. 65
5.5 Search-based approach VS Random: average values of the mutation score and number of configurations on the 30 runs. 65

6 A constraint-aware search approach for configuring large software product lines **72**
6.1 Distribution and evolution of the hypervolumes. 81
6.2 Impact of the violated constraints on the hypervolume for Linux. 84

7 Handling multiple testing objectives targeting a whole configuration suite **88**
7.1 Crossover operation 89
7.2 Mutation operation 90
7.3 Evolution of the objective function **F** and each normalized sub-objectives (to be minimized) during the 500 generations of the multi-objective approach. 94
7.4 Normalized sub-objectives (to be minimized) according to F_1 and F_2 comparison basis 95

7.5 Distribution of the p-values for the comparison with random 97

8 Assessing configurations with mutation and application to similarity testing 102
8.1 Mutation analysis approach for evaluating configurations. 103
8.2 Distribution of the 100 p-values resulting of the Mann-Whitney U Test between the mutation score achieved by similar and dissimilar configuration suites for the 100 executions. 106
8.3 Mutation score achieved for the prioritization techniques averaged on the 12 feature models. 108
8.4 Distribution of the 100 p-values resulting of the Mann-Whitney U Test between the mutation score achieved by configurations prioritized with the similarity technique and the 100 random prioritization. 108

9 Mutation analysis as a potential alternative to combinatorial interaction testing 112
9.1 The mutation analysis approach compared to the traditional combinatorial interaction testing. 113
9.2 Experimental methodology for comparing the mutation analysis approach with combinatorial interaction testing. 116
9.3 Distribution of the Kendall τ rank coefficients on different versions of the subject programs. 121
9.4 The easiness of finding faults, killing mutants and covering t-wise interactions per subject 122

10 From software product variants to a software product line: a preliminary approach 128
10.1 The ExtractorPL approach for the reverse-engineering of software product lines. . . 129
10.2 Example of three software product variants of a banking system. 130
10.3 Feature structure trees and construction primitives for the banking example. 131
10.4 The identified features for the banking software product variants. 132
10.5 The feature model built by ExtractorPL for the banking example. 132
10.6 The code units generated for each of the extracted features of the banking example. 133

11 Fixing re-engineered software product line feature models 142
11.1 Test-and-fix loop for feature models. 143

12 Testing card authorization systems with CETREL: a real-world case study 152
12.1 Process for testing the new card authorization systems in CETREL. 152
12.2 Approach proposed for improving the testing process prior testing 153
12.3 Structure of an authorization message. 154
12.4 Reduction and prioritization of 35,000 authorizations with 2-wise interaction coverage. 155
12.5 Merging daily traffic of authorizations to optimize the testing process 156

List of tables

4 Scalable t-wise generation and prioritization of software product line configurations 34

4.1 The 110 moderate size feature models involved in the empirical study. 40
4.2 The 4 large size feature models involved in the empirical study. 41
4.3 Comparison of the configurations' generation with ACTS, CASA and SPLCAT on the 10 smallest feature models of the empirical study for $t = 2, ..., 6$. 42
4.4 T-wise coverage (%) for the large feature models with 50 and 100 configurations. . . 46
4.5 6-wise coverage and fitness evolution over time for the search-based approach on the large feature models with 1,000 configurations. 46
4.6 Prioritization results: area under curve (scale 1:1,000). 49
4.7 Prioritization results: execution time in milliseconds. 50
4.8 Estimation of the interaction fault detection rate on the 4 large feature models for $t = 2, ..., 6$ for the search-based and unpredictable approaches. 54

5 A mutation-based approach for generating software product line configurations 58

5.1 The feature models used for the experiments. 62
5.2 Mutation operators used to alter feature models. 62
5.3 Comparison between the initial and final mutation score on the 30 runs for 1,000 generations. 63
5.4 Comparison between the search-based approach and a random one on the following basis: (a) same number of configurations and (b) same mutation score (MS) 64

6 A constraint-aware search approach for configuring large software product lines 72

6.1 Dissimilarity with and without Diversity Promotion (DP) on 1,000 configurations per feature model. 76
6.2 Feature models used in the empirical study. 77
6.3 Selected features in the seeds used by IBEA. 78
6.4 State-of-the-art VS the proposed approaches: comparison in terms of quality and diversity metrics on 30 independent runs per approach. 80

7 Handling multiple testing objectives targeting a whole configuration suite 88

7.1 Feature models used in the case study. 92
7.2 Evolution of the objective function **F** and the sub-objectives from the initial generation of the multi-objective approach to the final one (500 generations). 93
7.3 Comparison between the multi-objective approach and the random one 96

8 Assessing configurations with mutation and application to similarity testing 102

8.1 Mutation operators for feature models. 104
8.2 12 Various size feature models. 104
8.3 Mutation score achieved with different types of configuration suites (%). 105
8.4 Area under curve observed for the two prioritization techniques. 107

9 Mutation analysis as a potential alternative to combinatorial interaction testing 112

9.1 The four subjects programs used in the experiments. 119

10 From software product variants to a software product line: a preliminary approach 128

10.1 The notepad software product line used to evaluate ExtractorPL towards research question 1. 134

10.2 The software product lines used for the evaluation of the approach towards research question 2. 136

11 Fixing re-engineered software product line feature models 142

11.1 Evolution of the feature model problems over the repetitions (testing feedback) . . . 146

List of algorithms

4 Scalable t-wise generation and prioritization of software product line configurations **34**

1 Search-based configuration generation(m, t) 38

2 Local Maximum Distance prioritization (S) 39

3 Global Maximum Distance prioritization (S) 39

5 A mutation-based approach for generating software product line configurations **58**

4 Mutation-based generation of configurations 60

7 Handling multiple testing objectives targeting a whole configuration suite **88**

5 Multi-objective configuration suite generation 91

1

General introduction

In this chapter, we present the context and the challenges of this dissertation by setting out the general principles of highly configurable systems and software product lines, the associated terminology and the particularities leading to their increasing adoption by the software industry. We then describe the challenges related to the emergence of this kind of systems, and more specifically the issues addressed in this thesis.

Contents

1.1 Context **2**
1.1.1 Highly configurable systems and software product lines 2
1.1.2 Terminology 4
1.1.3 Increasing adoption of software product lines 4
1.2 Issues and contributions **4**
1.2.1 Challenges 5
1.2.2 Overview of the contributions and organization of the dissertation 6

1.1 Context

This dissertation focuses on the study of **highly configurable systems (HCSs)**, more particularly on the specific case of **software product lines (SPLs)**. HCSs are systems whose particularity is to propose many configuration options, leading to multiple variants of the same system, each of them varying according to the configuration options selected. SPLs are one type of HCSs [CDS07], whose characteristic is to allow the configuration and derivation of software products.

1.1.1 Highly configurable systems and software product lines

Configurations everywhere. Complex situations such as mixes of versatile environments, challenging user needs or time-to-market constraints are leading to the development of customizable and configurable software systems. The configuration of such systems is usually performed through a configuration interface where the user or the developer can set the different settings according to his preferences. These settings are usually predefined withing the system, intrinsically limiting the degree of variability and granularity proposed by the configurable piece of software. Systems proposing multiple configuration options are qualified as highly configurable.

Operating systems such as the Linux kernel form a good example of HCSs. Indeed, Linux can be configured to operate on a specific hardware, e.g., x86 or x64, but there is also the possibility to configure the functionalities of the system that will be available. For instance, the support for an internal webcam video capture tool can be enabled. Figure 1.1 shows an example of the configuration interface for the Linux kernel v2.2.16, which allows the user browsing the different menus to set up the system options. Constraints among the configuration options prevent the user from choosing incompatible settings, thus ensuring the consistency of the system. In line with the emergence of configurable systems, software development is increasingly moving from the production of a single, yet configurable software to the development of families of software products known as SPLs.

The concept of product line. Product line refers to the idea of designing a set of specific industrial products matching the needs of customers of a particular domain or market [KSP09, CN01]. The needs are expressed through features that can be proposed by the different products of the product

```
Linux Kernel v2.2.16 Configuration
                         Main Menu
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes,
<M> modularizes features. Press <Esc><Esc> to exit, <?> for Help.
Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

           Code maturity level options  --->
           Processor type and features  --->
           Loadable module support  --->
           General setup  --->
           Plug and Play support  --->
           Block devices  --->
           Networking options  --->
           Telephony Support  --->
           SCSI support  --->
           Network device support  --->
           Amateur Radio support  --->
           v(+)

              <Select>    < Exit >    < Help >
```

Figure 1.1: Configuration interface for the Linux kernel v2.2.6. The user can set the configuration options according to his preference prior the installation of the operating system.

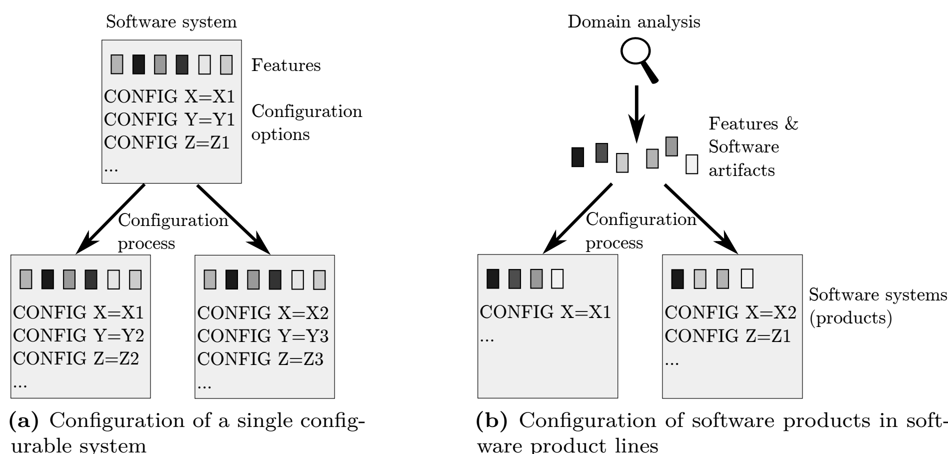
line. Some features may be common to all the products while some others may be specific. Once all the features are established, tailored products satisfying the specific requirements of the clients can be built, thus providing the market with a wide variety of products matching the customers expectations. As a concrete example, Figure 1.2a depicts a product line of smartphones. Some features are specific to all the phones, for instance, the display screen or the call function. On the contrary, some features are specific to each phone, such as the size of the screen or the presence or not of a front camera. The features bestowed by each smartphone result from a choice operated after a domain or market analysis, thus making each product designed to target a specific range of customers. The same principles have been applied to software engineering, leading to the concept of SPLs.

SPLs form a specific case of HCSs. The difference, which is illustrated in Figure 1.3, is that instead of having a single software system that can be configured in multiple ways and with predefined configuration options, SPLs allow to create tailored software products by combining the different features. Thus, the configuration step involves the selection of the different capabilities that will be offered and it results in a software product, which is itself a configurable software system. In SPLs, the products are software systems and the features are software functionalities or capabilities that are proposed by each software program. The variability among software products are expressed using these features [TBK09]. Like for HCSs, there are constraints governing the legal feature combinations. They are usually encoded in a feature model (FM), which is used to synthetically represent all the possible products of a SPL. Figure 1.2b depicts an example of the Bitdefender SPL. Each version proposes a common core of antivirus and Internet security functionalities. However, the "Total Security" version proposes unique features, i.e., that are not included in the two other editions, such as file encryption.

(a) A product line of smartphones

(b) The Bitdefender software product line

Figure 1.2: Example of (software) product line.



(a) Configuration of a single configurable system

(b) Configuration of software products in software product lines

Figure 1.3: Simplified view of the configuration process in configurable systems and software product lines. Configuring a configurable system results in the same system. In software product line, the configuration process uses features to construct tailored software products.

1.1.2 Terminology

To avoid confusion, we define the recurring terminology used throughout the dissertation. Further details regarding the following concepts and additional definitions are provided in Chapter 2.

- A **feature** denotes a functionality or characteristic of a software product of a SPL [CHS08].
- A **configuration** denotes the list of features that are combined in a given software product of a SPL.
- **Configuration space** refers to the set of all the possible product configurations of a SPL.
- A **Software product (SP)** denotes the result of the configuration process in SPLs.
- **Software product variants (SPVs)** are a set of SPs which share common features.
- **Constraints** between features are rules which, when they are all simultaneously fulfilled, makes a a configuration **valid**. When one or more constraint is not satisfied, the configuration is said to be **invalid**.
- **Feature models** are used to represent a SPL. They encode the features of the SPL and their associated constraints.
- Configuration **generation** denotes the process of providing a set of valid configurations for testing purposes.
- Configuration **prioritization** refers to the process of ordering a given set of valid configurations for testing purposes.

1.1.3 Increasing adoption of software product lines

Software reuse involves generating new designs by combining high-level specifications and existing component artifacts [KSP09]. Reusability and modularity is at the heart of the SPL methodology. The reusability principles have always been part of software development, with the emergence of modules in the 1970's and 20 years later with objected-oriented programming. Software development paradigms have thus constantly been moving towards reusing the software components. Pushing further the concept of reusability, the trend nowadays is to reuse pieces of software which can contain many components or objects, defining the notion of feature. Managing, combining and reusing features is the essence of SPL engineering. The benefit of this paradigm includes a flexible productivity, higher quality in the developed SPs, reduced cost and a faster time to market. However, SPLs introduce several challenges.

1.2 Issues and contributions

The spreading of SPLs and their adoption by the industry has revealed several issues related to SPL testing. This section first presents the major challenges that we identified and it then describes the problems addressed in this dissertation. To this respect, Figure 1.4 depicts an overview of these issues, which are discussed in the following.

1.2.1 Challenges

Millions of configurations. Testing a SPL is an inherently difficult activity [McG10]. Although testing all the products that can be configured would be ideal, it is rarely feasible in practice. Indeed, the number of possible configurations induced by a SPL usually grows exponentially with the number of features, quickly leading to millions of possible products to test. Since in a real world and industrial environment, the resources are limited by budget and time constraints, test engineers are seeking solutions to reduce the size of their test suites, i.e., the products to test, so that they can meet release deadlines and cost constraints. In this respect, testing techniques such as combinatorial interaction testing (CIT) have been proposed to reduce the size of the test suites, but they face scalability issues.

The limitations of existing CIT techniques. CIT reduces the size of the test suites using interaction coverage. It is a systematic approach for sampling large domains of test data. It is based on the observation that most of the faults are triggered by the interactions between a small number of features [KWG04]. An interaction between $t \geq 2$ features denotes the possible impact of one feature on the others, enabled in a specific configuration. While this technique has been widely used in the area and proven to be effective (for instance, Kuhn *et al.* [KWG04] have shown that interactions between two features are able to disclose 80% of the bugs) existing tool face scalability issues, especially when there are many constraints ruling the possible combination between the features. As a result, they are often limited to interactions between 2 or 3 features and SPLs of about 1,000 features, but real SPLs can go over 6,000 features and may require higher interaction strength [KLK08, NKN14]. In addition, most of the approaches do not consider prioritizing the resulting configurations.

The lack of multiple testing objectives. In real-life situations, generating configurations to test is a multi-dimensional problem. Indeed, the testing process usually involves multiple objectives. Testing products with respect to CIT may be one objective, but we might want at the same time to minimize the number of configurations, to maximize the number of features proposed and to minimize the cost of testing these products. Such requirements necessitate a trade-off between several testing objectives.

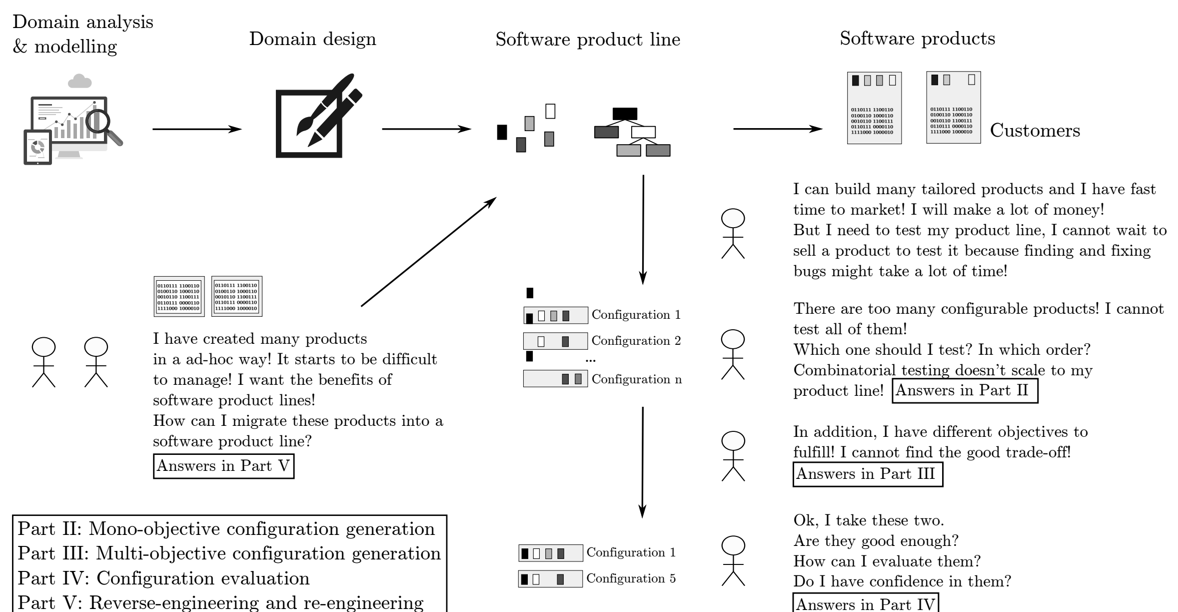


Figure 1.4: Issues addressed in the dissertation.

Without automated support, the configuration process is likely to be highly suboptimal: it requires the simultaneous satisfaction of multiple and possibly conflicting objectives, i.e., the maximization of one objective can penalize another objective. Most of the approaches for generating configurations to be tested target a single objective at a time. While this may be sufficient in certain cases, this method does not reflect real-life testing situations and constraints.

The need for evaluating configurations prior to testing. Despite the generation of configurations according to a single or to multiple testing objectives, one issue is to evaluate the quality of a given set of configurations prior to the testing process. Since testing SPs can take several hours or days, thus leading to important testing costs, it is important to be able to evaluate the quality of such configuration upstream the testing process. Generally, CIT serves as a yardstick towards assessing the ability of a given set of configurations to reveal faults. In other words, the combinatorial testing criterion forms a measurement of the test suite effectiveness. However, calculating the interaction coverage is computationally expensive, leading to the necessity to find alternatives to this criterion. Thus, there is a need to investigate alternative configuration evaluation methods which are at least as effective in terms of fault detection and faster to compute than combinatorial testing.

Transforming software product variants to software product lines. Migrating existing SPVs to a a SPL is a challenging task whose profit include a reduced risk to introduce errors when new products are created. Indeed, creating SPVs with ad-hoc techniques like copy-paste-modify can lead to an introduction of errors in some SPVs. If the code of each feature is centralized within the SPL and shared in all the products proposing these features, it allows testing each feature independently. This allows using SPL testing techniques which aim at testing the whole SPL in an efficient way. Thus, testing product SPVs is simpler if they are migrated into a SPL via an automated reverse-engineering approach. However, there are few of such approaches and, like any reverse-engineering technique, they are error-prone. As a result, techniques for automatically evaluating and fixing reverse-engineered SPLs and FMs are needed so that they adequately represent the software systems they are based on.

1.2.2 Overview of the contributions and organization of the dissertation

This dissertation addresses the above-mentioned challenges by using **search-based (SB)** approaches [HMZ12] combined with constraint solvers. SB approaches are particularly profitable in the SPL context since the configuration space can involve millions of configurations and can thus not be exhaustively searched, and the use of solvers allows us to deal only with valid configurations. Indeed, invalid configurations are useless from a practical point of view as they lead to SPs that cannot be deployed.

This thesis encompasses four main parts, each one proposing solutions to address the challenges presented in the previous section. These four parts are preceded by a first one introducing the technical background of this dissertation and a state of the art, and they are followed by a last part presenting an industrial case study and a general conclusion. In the following, the structure of the dissertation along with an overview of the contributions of each part are presented and depicted in Figure 1.5.

Part I: Background and state of the art. In this part, technical background of this dissertation and definitions of the concepts used are introduced in Chapter 2. Related work is discussed in the following chapter, Chapter 3.

Part II: Mono-objective configuration generation. Chapter 4 introduces scalable techniques for both generating and prioritizing configurations with respect to the CIT criterion. The presented approaches outperforms the state of the art and allows reaching high interaction strengths by using a SB approach relying on a similarity heuristic and combined with constraint solvers. An empirical study conducted on 114 SPLs demonstrate the feasibility and practicality of the introduced techniques. In Chapter 5, an approach relying on an alternative criterion to the CIT one, the mutation criterion, is used in a SB approach for generating configurations. Mutation promotes configurations revealing faulty versions of the FM.

Part III: Multi-objective configuration generation. While Chapters 4 and 5 sticks to a single objective for generation configurations, Chapter 6 introduces a multi-objective generation approach for generating SPL configurations satisfying multiple testing objectives. The technique enhances and outperform the sate of the art by adding constraint solving support. While these testing objectives target a single configuration, the following chapter generalizes the multi-objective approach by introducing the first approach supporting testing objectives targeting a set of configurations. The approaches introduced in this part have been validated through case studies.

Part IV: Configuration evaluation. In this part, Chapter 8 uses the mutation criterion introduced in Chapter 5 to evaluate the quality of configurations. A link is established between the mutation criterion and the similarity heuristic presented in Chapter 4. In particular, it is shown that the similarity heuristic is good to detect faulty versions of the FM. The following chapter, Chapter 9 shows that this alternative criterion, mutation, correlate with fault detection and that it can form a viable alternative to CIT. These two chapters also conduct empirical studies to demonstrate their practicality.

Part V: Reverse-engineering and re-engineering. Chapter 10 presents a preliminary automated and language-independent approach for migrating SPVs into a SPL. It is the first approach which is fully automated. The next chapter introduces a test and fix loop which uses search to automatically test and fix reverse-engineered FMs. The idea behind this approach is to automatically improve reverse-engineered FMs so that they represent better the system they model. Fixing such models is important as all the previous parts operate on the SPL FMs. Both these chapter conduct case studies to assess their benefits.

Part VI: Industrial application and final remarks. In this part, Chapter 12 presents an industrial application of the techniques presented in Parts II, III and IV with the CETREL company. CETREL is a company handling credit card transactions in Luxembourg. We optimize the testing process of their card authorization system by modeling credit card authorizations as a SPL FM, enabling the application of the above-mentioned generation and evaluation techniques. In particular, the redundancy among authorizations is removed, reducing the authorizations by more than 80%, and variability metrics are provided to assess the quality of the authorizations. Finally, Chapter 13 concludes the dissertation and discusses future research directions.

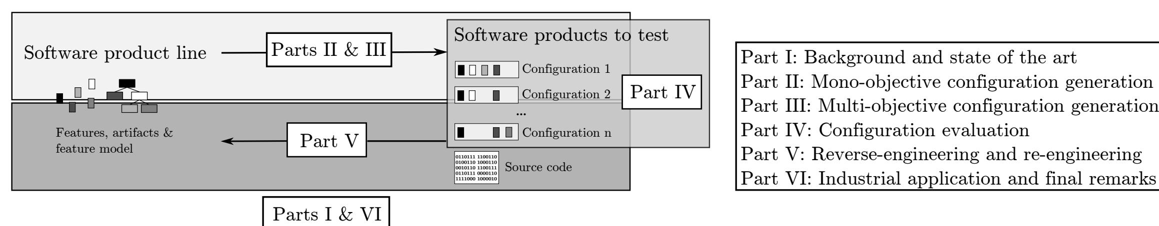


Figure 1.5: Structure of the dissertation.

Part I

Background and state of the art

2

Technical background and definitions

This chapter presents technical background and definitions used in this dissertation.

Contents

2.1 Model-based software product lines **12**
2.1.1 Software product line engineering 12
2.1.2 Feature models 12
2.1.3 Configurations 13
2.1.4 Mutation analysis 15
2.2 Combinatorial interaction testing **15**
2.2.1 T-wise testing and coverage 16
2.2.2 The impact of constraints 16
2.2.3 Combinatorial interaction testing models 17
2.3 Optimization problems **17**
2.3.1 Metaheuristics 17
2.3.2 Search-based software engineering 17
2.3.3 Multi-objective optimization 19
2.3.4 Configuration generation and prioritization 19

2.1 Model-based software product lines

2.1.1 Software product line engineering

Software engineers build many variations of their systems in order to match the specific needs of particular clients [CHW98]. **Software product line engineering (SPLE)** [CN01] is a software development paradigm designed to handle this situation. It applies product lines techniques to software systems and involves the creation and the management of a SPL which encompasses the different variants, called products. SPLE appeared in 1990 with the development of feature-oriented domain analysis [KLD02]. SPLE handles variability as a first-class concept through **feature models (FMs)** [KCH+90].

2.1.2 Feature models

In essence, a FM aims at defining legal combinations of features authorized or supported by a system using hierarchical decomposition and additional constraints. This dissertation focuses on model-based testing of SPLs where the variability model is a FM. Such a model encompasses the constraints linking the features, thus defining the legal combinations between them. Feature modeling is a popular way to model SPL variability and it is by far the most reported in industry [BRN+13]. Thus, basing SPL testing techniques on FMs as means of documenting variability seems appropriate. Moreover, FMs may be used to reason about systems that are not SPLs according to the classical definitions [CN01, PBL05]. Thus, a FM can represent the variability of a SPL or of a highly configurable system.

Definition 1 (Feature model) *Let a FM be a tuple $FM = (F, K)$, where $F = \{f_1, ..., f_n\}$ is set of n Boolean features and $K = \{c_1, ..., c_k\}$ a set of k constraints over the features. A constraint c is satisfied if all its constraints are satisfied, i.e., evaluated to true.*

As an example, consider the FM depicted in Figure 2.1. It contains 9 features: $F = \{f_1, ..., f_9\}$. Some features are mandatory (included in every product), e.g., the "draw" feature. Other features

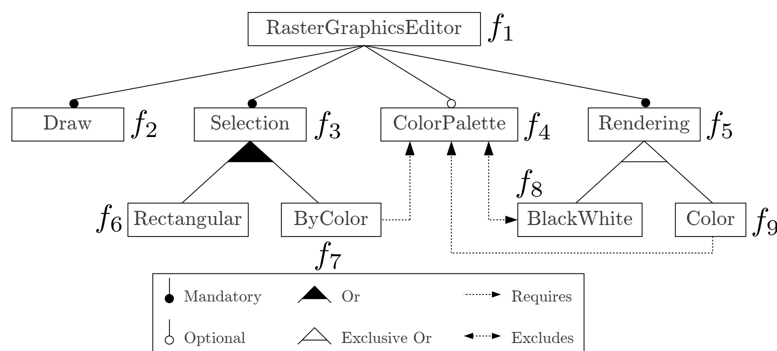


Figure 2.1: A feature model of a raster graphics editor software product lines. It encompasses the 9 features and the constraints linking them.

are constrained in their occurrence. For instance, the "color" feature requires the "color palette". It also encompasses $k = 18$ constraints represented as follows in conjunctive normal form:

$K = \{f_1, (\overline{f_2} \vee f_1), (\overline{f_1} \vee f_2), (\overline{f_3} \vee f_1), (\overline{f_1} \vee f_3), (\overline{f_4} \vee f_1), (\overline{f_5} \vee f_1), (\overline{f_1} \vee f_5), (\overline{f_6} \vee f_3), (\overline{f_7} \vee f_3), (\overline{f_3} \vee f_6 \vee f_7), (\overline{f_8} \vee f_5), (\overline{f_9} \vee f_5), (\overline{f_5} \vee f_8 \vee f_9), (\overline{f_8} \vee \overline{f_9}), (\overline{f_7} \vee f_4), (\overline{f_4} \vee \overline{f_8}), (\overline{f_9} \vee f_4)\}$.

A FM can also be converted to a Boolean formula [MWC09] to be used within constraint solvers, such as **satisfiability (SAT)** solvers [LBP10] for reasoning and analysis. Such Boolean formulas are a conjunction of logical clauses, where a clause is a disjunction of m literals. As a result a clause represents a constraint of the FM and a literal is a feature that is selected (f_j) or not ($\overline{f_j}$). Thus, a FM can also be written in the form of a Boolean formula as follows:

$$FM = \bigwedge_{i=1}^{k} \left(\bigvee_{j \in [1;n]} l_j \right), \text{ where } l_j = f_j \text{ or } \overline{f_j}.$$

For instance, the corresponding Boolean formula of the FM of Figure 2.1 is a conjunction of all the constraints:

$FM = f_1 \wedge (\overline{f_2} \vee f_1) \wedge (\overline{f_1} \vee f_2) \wedge (\overline{f_3} \vee f_1) \wedge (\overline{f_1} \vee f_3) \wedge (\overline{f_4} \vee f_1) \wedge (\overline{f_5} \vee f_1) \wedge (\overline{f_1} \vee f_5) \wedge (\overline{f_6} \vee f_3) \wedge (\overline{f_7} \vee f_3) \wedge (\overline{f_3} \vee f_6 \vee f_7) \wedge (\overline{f_8} \vee f_5) \wedge (\overline{f_9} \vee f_5) \wedge (\overline{f_5} \vee f_8 \vee f_9) \wedge (\overline{f_8} \vee \overline{f_9}) \wedge (\overline{f_7} \vee f_4) \wedge (\overline{f_4} \vee \overline{f_8}) \wedge (\overline{f_9} \vee f_4)$.

Feature models are now equipped with formal semantics [SHT06], automated reasoning operations and benchmarks [BSRC10], tools [KTS+09] and languages [Bat05]. There are also used to derive software products [BSRC10], configure them [AHH11] and for automated quality assurance [POS+12].

Finally, while this dissertation focuses on FMs to represent the SPL variability, other formalisms may be used to represent variability. For instance, Acher *et al.* [ABBJ14] used openSCAD, a non-visual modeling tool for 3D printing. There is also the unified modeling language (UML) [ZJ06], which can combined with the object constraint language (OCL) to express the feature constraints [ZHJ04]. FMs were also declined to represent other things than the possible software products that can be configured, such as the marketing view or the environmental context [CCMJ12] or a taxonomy of model transformation [CH06].

2.1.3 Configurations

Configurations represent the features that are proposed by a software product of a SPL. In this context of SPLs based on FMs, let a configuration be an assignment of selected/unselected features satisfying the constraints of the FM.

Definition 2 (Configuration) *A configuration (of a software product of a SPL) is a set $C = \{\pm f_1, ..., \pm f_n\}$, where $+f_i$ indicates a feature of the FM which is present in the corresponding software product, and $-f_i$ an absent one. A configuration is said to be **valid** if K is satisfied, i.e., all the constraints of the FM are simultaneously satisfied. Otherwise, it is said to be **invalid**. When a configuration is valid, we also that it **satisfies the FM**.*

For instance, with respect to Figure 2.1, $C = \{+f_1, +f_2, +f_3, +f_4, +f_5, -f_6, +f_7, -f_8, +f_9\}$ is the configuration of the software product proposing all the features except rectangular selection and black and white rendering. This configuration is valid since it satisfies the constraints of the FM described in the previous subsection. On the contrary, $C' = \{+f_1, -f_2, +f_3, +f_4, +f_5, -f_6, +f_7, -f_8, +f_9\}$ violates the constraint $(\overline{f_1} \vee f_2)$ and is thus invalid.

A configuration may actually refer to different things depending on the source of the features. While in the SPL terminology it usually refers to the configuration of a software product of the SPL, where

each of its features may have a complex behavior, it may as well represent parameter values if the FM is a configurable system, e.g., the Linux kernel. This is not a problem in our case since the approaches described in this dissertation agnostic of feature semantics [CHS08]. Regarding the SPL testing process, these configurations need to be embodied and relevant test cases for these configurations have be provided so that actual testing can be performed. The full process is out of the scope of this dissertation. Nevertheless, model-based product configuration generation techniques show that this scenario is realistic even for large systems [JHF+12c].

2.1.3.1 On the validity of configurations

To decide whether a configuration is valid or invalid (with respect to the constraints of the FM), the Boolean formula of the FM is encoded into a SAT solver. Since the formula is a conjunction between all the constraints,this formula can be evaluated to *true* only when all the constraints are satisfied. Thus, since a configuration is the list of selected (*true*) and unselected (*false*) features, it represents a specific assignment of the formula variables. A configuration is *valid* when the formula is evaluated to *true* by the constraint solver.

Unless specified using the terms valid and invalid, the remainder of this dissertation uses the terminology configuration to refer to a valid configuration. Indeed, from a practical testing perspective, invalid configurations are useless in practice as they may lead to faulty software products.

2.1.3.2 Configuration suite

We finally denote as configuration suite a set of configurations. Such configuration suites are the results expected from the generation approaches presented in Parts I and II.

Definition 3 (Configuration suite) *A configuration suite is a set $CS = \{C_1, ..., C_m\}$ where each C_i is a valid configuration and where the order of the configurations is not important.*

2.1.3.3 Modeling the testing cost of configurations

In an attempt to take into account the testing cost of configurations, some assumptions are made. We assume that the testing effort of each configuration is related to the number of features that it contains. Additionally, each feature requires a different amount of resources in order to be tested. To this end, a value representing an estimate of its testing cost is assigned to each feature. Thus, the cost of testing one configuration is assumed to be equal to the sum of the cost of the features that it is composed of. More formally, if $cost(f_i)$ denotes the cost of the feature f_i, the a configuration $C = \{\pm f_1, ..., \pm f_n\}$ has a cost *cost* equals to:

$$cost(C) = \sum_{i=1}^{n} p(i)cost(i), \text{ where } p(i) = \begin{cases} 1 & \text{if } +f_i \\ 0 & \text{if } -f_i \end{cases}.$$

The cost of a test configuration suite is the sum of the cost of the m configurations that it is composed of. Thus, the cost of $CS = \{C_1, ..., C_m\}$ is given by:

$$cost(CS) = \sum_{i=1}^{m} cost(C_i).$$

Putting a cost to each feature transforms the initial FM into an attributed FM [OSCR12]. A similar way of representing the cost of features and configurations can be found in the work of Olaechea *et al.* [OSCR12].

2.1.4 Mutation analysis

Mutation analysis forms a powerful technique with various applications like software testing [JH11, Off11] and debugging [PLT12]. It is applied by creating altered (mutant) versions of the various programs' artifacts like source code, specification models, etc. [JH11, Off11]. The main idea behind this approach is to evaluate the ability of test cases to reveal behavior differences between the original (unaltered) and the mutated (altered) artifact versions. The mutated versions represent possible defects of the artifact under test and they are produced based on a set of well defined rules called mutation operators [Off11].

Mutation operators are defined on "syntactic descriptions to make syntactic changes to the syntax or objects developed from the syntax" [Off11]. The process of introducing mutants is called mutation analysis. The ability of the utilized test cases to reveal the introduced mutants is examined in order to use this approach for testing purposes (mutation testing). If a mutant can be detected by a test, the mutant is called killed. Otherwise, it is called live. Therefore, measuring the ratio of the killed mutants to the totally introduced ones results in a quality measure of the testing process. This measure is called **mutation score (MS)** and demonstrates the ability of the tests to detect errors.

In the context of this dissertation, mutants are produced by applying a set of mutation operators to the original FM. Such operators alter the logical constraints of the FM. The test evaluation is performed by checking whether a configuration is valid towards the modified FMs' constraints, i.e., whether the modified Boolean formula is evaluated to *true*. Since the examined configurations are produced based on the original FM, they always satisfy their respective Boolean formulas. Consequently, a mutant is said to be killed if its formula is not satisfied, i.e. if the formula is evaluated to *false*.

2.2 Combinatorial interaction testing

In testing, CIT aims at sampling test suites in order to reduce the testing effort. This reduction is achieved by keeping only the test cases covering all the interactions between t parameters. The level of interaction or strength is generally denoted as t. T-wise testing refers to the process of applying CIT with a strength $t \geq 2$.

CIT approaches are closely related to the configuration generation in SPLs. Generally, CIT handles multi-valued variables while configuration generation for SPLs limits the values of variables, i.e., the features, to Boolean ones. In that sense, the generation of configurations in SPLs can be seen as a subset of CIT. However, constraints among variables are generally not included in CIT problems. Indeed, initial CIT approaches did not take into account constraints. Some recent studies, e.g., [CDS07, Pet15] evaluate the impact of constraints on CIT. Recent CIT tools provide a support of constraints. In other words, we can say that configuration generation in SPL testing is an instance of a Boolean CIT problem with constraints. The following mapping between the terminology can be used: test suites of CIT are configuration suites and the parameters of CIT are the features of SPLs.

Generally, a SPL configuration generation problem can be solved by a CIT tool if it handles constraints. Indeed, CIT with constraints generalizes the SPL configuration generation problem since it deals

with multi-valued variables. However, to convert a CIT problem to a SPL one, it is necessary to transform the problem into a Boolean one. If the variables' domain is finite, this transformation can be performed by applying the rules presented by Frisch *et al.* [FPDN05].

2.2.1 T-wise testing and coverage

T-wise testing focuses on the interactions between any $t \geq 2$ features of a SPL [PSK+10]. It considers all the possible interactions (with respect to the constraints K of the FM) between selected and unselected features. Such an interaction is called a t-set.

Definition 4 (valid t-set) *A valid t-set is a set $\{\pm f_1, ..., \pm f_t\}$ satisfying K, with $t \leq n$ and where $+f_i$ indicates a feature which is selected and $-f_i$ an unselected one. A t-set which is not satisfying K is said to be invalid.*

As a result, a t-set can be seen as a partial configuration, and their validity or invalidity is evaluated in a similar way as previously described in Section 2.1.3.1.

Definition 5 (t-wise coverage) *The t-wise coverage of a configuration suite $CS = C_1, ..., C_m$ is defined by the following ratio: $\frac{\# \bigcup_{i=1}^{m} T_{t,C_i}}{\# T_{t,FM}}$, where T_{t,C_i} is the set of t-sets covered by the configuration C_i (i.e., t-sets included within the configuration C_i), where $T_{t,FM}$ denotes the set of all the possible t-sets of the FM and where $\#A$ denotes the cardinality of the set A.*

Classical approaches [OMR10, POS+12, JHF11] to t-wise testing have coverage ratio of 1 as they cover all the t-sets of the FM. Finally, set coverage redundancy expresses the possibility that, by removing any configuration, the coverage value is not altered.

As an example, consider the FM depicted in Figure 2.1 and the following configurations:

$$
\begin{aligned}
C_1 &= \{+f_1, +f_2, +f_3, +f_4, +f_5, -f_6, +f_7, -f_8, +f_9\},\\
C_2 &= \{+f_1, +f_2, +f_3, +f_4, +f_5, +f_6, -f_7, -f_8, +f_9\},\\
C_3 &= \{+f_1, +f_2, +f_3, +f_4, +f_5, +f_6, +f_7, -f_8, +f_9\},\\
C_4 &= \{+f_1, +f_2, +f_3, -f_4, +f_5, +f_6, -f_7, +f_8, -f_9\}.
\end{aligned}
$$

The t-sets of this FM can be computed as follows. Compute all the possible t combinations from $\{+f_1, ..., +f_9, -f_1, ..., -f_9\}$. Then, remove the combinations that are invalid. An example of valid 3-set is $\{+f_1, +f_2, -f_8\}$. An invalid 2-set is for instance $\{-f_1, f_2\}$, as it does not satisfies the constraint $-f_2 \vee +f_1$. Thus, the example FM encompasses 73 valid 2-sets, 204 valid 3-sets, etc. C_1 and C_4 together cover $66/73 \approx 90.4\%$ of these 2-sets and $\approx 80.4\%$ of the 3-sets. On the contrary, C_2 and C_3 together cover only $\approx 60.3\%$ of the 2-sets and $\approx 54.9\%$ of the 3-sets.

2.2.2 The impact of constraints

Constraints have a great influence: they can, as described in the previous sections, make configurations invalid. They may also increase or decrease the number of configurations required to cover all the t-wise interactions, or produce invalid t-sets [CDS07]. Thus, introducing constraints into a CIT

problem makes it extremely difficult to solve. This is due to the irregularity introduced by the constraints [CDS07]. The configuration generation in SPLs suffers from the same problems.

As an example, consider the SPL depicted by the FM of Figure 2.1. In the absence of constraints, $2^9 = 512$ configurations can be established (9 features, with 2 possible values per feature). However, there are configurations among these 512 that are invalid with respect to the constraints. Taking into accounts the constraints drops the number of configurations to 4 only (i.e., this SPL supports only 4 configurations). In other words, only 4 configurations among the 512 possible are valid ones. It means that, by trying to randomly generate a configuration, the probability to obtain a valid one is only $4/512 = 0.78\%$. As a result, generating valid configurations at random is rather unlikely, even for small SPLs. Thus, a systematic way to deal with valid configurations is in need. To deal with this situation, a SAT solver [LBP10] to handle the constraints of the FM is mandatory.

2.2.3 Combinatorial interaction testing models

CIT models can be seen as a generalization of feature models, where the variables can take multiple values from a finite set instead of Boolean ones. In this dissertation, we consider CIT models with constraints expressed in a similar ways as those of FMs.

2.3 Optimization problems

Optimization problems fall into two categories: problems that can be solved with an exact technique in a reasonable amount of time, and the others, which require the introduction of metaheuristics to find near-optimal solutions.

2.3.1 Metaheuristics

Metaheuristics are used when the solution space is extremely large, making it practically impossible to evaluate all the solutions to find the best one, and where there is no known efficient technique to find the exact best solution. The general idea is, in such cases, to find approximate solutions by defining a way to decide which potential solution is the best among two. Two potential solutions are in this case compared in terms of **fitness**, noted f, and the objective is to find the best solutions according to that fitness.

2.3.2 Search-based software engineering

Search-based software engineering (SBSE) [HMZ12] denotes a family of techniques which convert a software engineering problem into a computational search problem which can be tackled with a metaheuristic. One of the most famous metaheuristic category in SSBSE is evolutionary algorithms, which encompasses genetic algorithms.

2.3.2.1 Genetic algorithms

Genetic algorithms form search-based heuristics mimicking the natural evolution process. They represent a smart way to randomly search for solutions to optimization problems. To apply such an approach, several parameters like the genes, the individuals and the fitness function have to be defined. The individuals correspond to what composes a possible solution to the optimization problem. Each individual is composed of several units, called genes and the set of individuals that is handled by the algorithm is called the population. The fitness function quantifies the individuals' ability to solve the optimization problem. Generally, these algorithms operate by repeatedly reproducing, adjusting and selecting the best individuals of the population. Based on the process presented in the following subsection, the population is gradually evolved by optimizing the solutions it encodes.

The process. Genetic algorithms operate by evolving a population of potential solutions called individuals. The evolution is guided by a fitness function. The initial population is usually produced at random and evolved based on a given set of operations on its individuals. Usually, three operations are used for the evolution of the population. These are the selection, crossover and mutation [HM10].

1. *Selection* chooses individuals for performing crossover and mutation. The selection is made by choosing the individuals with the best scores according to the fitness function.

2. *Crossover* selects two individuals and switches some of their genes. This is usually performed by ordering the individuals' genes and switching all the genes after a randomly selected point. Crossover results into two new individuals called offsprings.

3. *Mutation* performs on an offspring by changing the values of one or more of its genes.

Performing the selection, crossover and mutation operations on a population results in one evolution cycle of the population. This cycle is called population generation. At each generation, the individuals that fit the best to the problem, i.e., which have the best fitness, are kept. This principle is adapted from the Darwin's theory of evolution, i.e., survival of the fittest. An overview of one generation is presented in Figure 2.2. The algorithm terminates after completing a predefined number of generations.

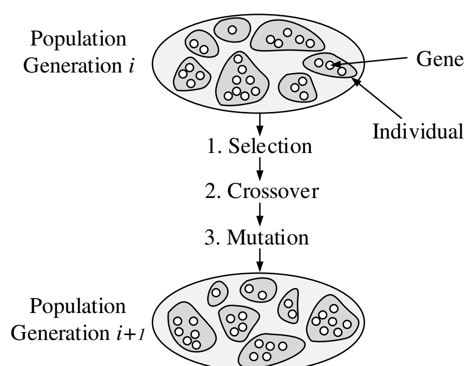


Figure 2.2: The process of evolving a population in genetic algorithms. The current population is evolved into a new one by selecting, crossing and mutating individuals.

2.3.3 Multi-objective optimization

Multi-objective optimization (MOO) refers to the process of optimizing more than one objective at the same time. The aim of these approaches is to search for optimal (or nearly optimal) solutions requiring trade-offs between two or more conflicting objectives.

Let X be the set of all the possible solutions to a problem and let $\mathbf{F} = [F_1(x), ..., F_k(x)]^T$ be a vector of k objective functions. If each objective has to be minimized, MOO aims at finding $x_1, ..., x_k$, i.e., the solutions to the problem, such as $\mathbf{F}$ is minimized. The minimization of $\mathbf{F}$ is the process of optimizing simultaneously the k objective functions [MA04]. The terms objective function and fitness function are similar, but generally fitness is used for single-objective optimization while objective is used for MOO problems. Finally, multi-objective evolutionary algorithms denotes a set of evolutionary techniques used to solve MOO problems.

2.3.3.1 Pareto front

Let x_1 and x_2 be two potential solutions to a MOO problem. We say that x_1 dominates x_2, written as $x_1 \succ x_2$ if and only if $\forall i \in \{1, ..., k\}\, F_i(x_1) \leq F_i(x_2)$ and $\exists i \in \{1, ..., k\}\, F_i(x_1) < F_i(x_2)$. Given $x_1, ..., x_n$ potential solutions to the MOO problem, the **Pareto front (PF)** corresponds to the subset of these potential solutions that are non-dominated by the others.

An example of PF is illustrated by Figure 2.3 for two objectives F_1 and F_2 to minimize. In this example, x_1, x_2, x_4, x_6 and x_7 are in the PF set since they are not dominated by any other solution. By contrast, x_{10} is dominated (among others) by x_2 ($x_2 \succ x_{10}$). So, it does not lie on the front. Finally, we denote as PF size the number of solutions in the PF.

2.3.4 Configuration generation and prioritization

This section defines configuration generation and prioritization as used in this dissertation.

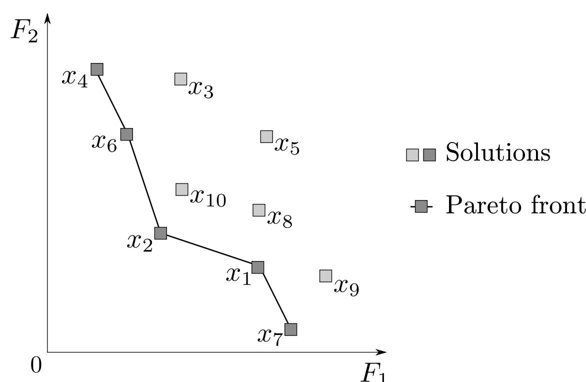


Figure 2.3: An example of Pareto front with two objectives F_1 and F_2 to minimize. The solutions x_1, x_2, x_4, x_6 and x_7 are in the Pareto front since they are not dominated by any other solution.

Definition 6 (Configuration generation) *The objective of the generation process is to provide a configuration suite that fulfills the requirements of a test criterion. If X denotes the set of all the configurations (i.e., solutions) and $CS = \{C_1, ..., C_m\}$ a configuration suite, this process is formally defined as:*

- *Given: a FM, the desired number of configurations, m, a given amount of time, a_t, and a fitness function f from CS to the real numbers, $f : C^m \longrightarrow \mathbb{R}_+$.*
- *Problem: finding $CS \in C^m$ with respect to a_t such as $[max(f(CS))]$.*

In the present context, the testing criterion is modeled by a fitness function f. In this dissertation, we will use the t-wise coverage or the mutation score as test criteria.

Definition 7 (Configuration prioritization) *The aim of this process is to order a configuration suite $CS = \{C_1, ..., C_m\}$ according to their ability to fulfill the test criterion. Therefore, by testing $k \leq m$ configurations, the greatest possible level of f, for any number of k configurations, is achieved. More formally [YH12]:*

- *Given: a configuration suite, CS, the set of all the permutations of CS, P_{CS} and a fitness function f from P_{CS} to the real numbers, $f : P_{CS} \longrightarrow \mathbb{R}_+$.*
- *Problem: finding $CS' \in P_{CS}$ such as $(\forall CS'' \in P_{CS} \mid CS'' \neq CS')[f(CS') \geq f(CS'')]$.*

In this dissertation, the testing criterion model by f for prioritization is the t-wise coverage.

As an example, consider the FM of Figure 2.1 and the following configurations:

$$
\begin{aligned}
C_1 &= \{+f_1, +f_2, +f_3, +f_4, +f_5, -f_6, +f_7, -f_8, +f_9\},\\
C_2 &= \{+f_1, +f_2, +f_3, +f_4, +f_5, +f_6, -f_7, -f_8, +f_9\},\\
C_3 &= \{+f_1, +f_2, +f_3, +f_4, +f_5, +f_6, +f_7, -f_8, +f_9\},\\
C_4 &= \{+f_1, +f_2, +f_3, -f_4, +f_5, +f_6, -f_7, +f_8, -f_9\}.
\end{aligned}
$$

In this example, if $m = 2$ (i.e., 2 configurations have to be generated), we expect them to be those that provide the maximum coverage. Therefore, with respect to Section 2.2.1, C_1 and C_4 should be chosen rather than C_2 and C_3, as they cover more t-sets.

Suppose now that we have only the three configurations C_1, C_2 and C_3 that we want to prioritize. All together, they provide a 2-wise coverage of $\approx 71.2\%$. C_1 alone provides a 2-wise coverage of $\approx 49.3\%$. If we now consider C_2 in addition to C_1, it increases the coverage to $\approx 69.8\%$. However, if we consider C_3 in addition to C_1, the coverage is extended to only $\approx 60.2\%$. This difference is depicted in Figure 2.4. In other words, the order in which the configurations are considered allows reaching faster or slower the total coverage of $\approx 71.2\%$ provided by these configurations. In this case, it is more interesting to consider the order C_1, C_2 and C_3 rather than the order C_1, C_3 and C_2.

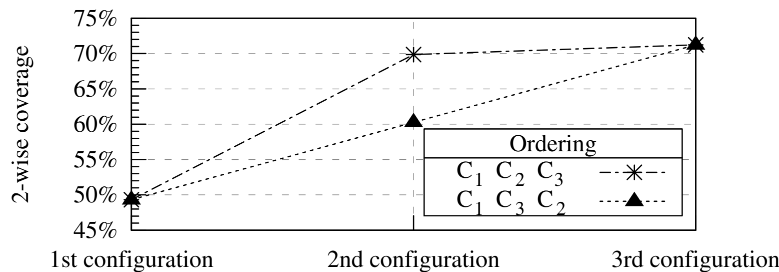


Figure 2.4: A different ordering of the configurations allows reaching faster or slower the t-wise coverage provided by these configurations.

Area under curve. The experiments conducted in this dissertation for evaluating prioritization techniques make use of the area under curve notion. According to Do and Rothermel [DR06], "the area under the curve represents the weighted average of the percentage of faults detected over the life of the test suite. This area is the prioritized test suite's average percentage faults detected metric ." In this dissertation, we measure the t-wise coverage instead of the percentage of faults. Hence, the area under curve represents the effectiveness of the studied approaches. This area is the numerical approximation of the integral of the discrete coverage curve and is computed using the trapezoidal rule, i.e., $\int_a^b g(x)dx \approx (b-a)\frac{g(a)+g(b)}{2}$. Thus, for each prioritization method, if $cov(x)$ denotes the percentage of t-wise coverage achieved with the x-th configuration, then the area value is given by $\sum_{i=1}^{99} \frac{cov(i)+cov(i+1)}{2}$. A higher area under curve value expresses a more effective prioritization.

For instance, with reference to Figure 2.4, we can see that the area below the ordering C_1, C_2 and C_3 is bigger than the area below C_1, C_3 and C_2, thus indicating a more effective prioritization.

3

Related work

This chapter discusses work related to the one presented in this dissertation.

Contents

3.1 Configuration generation and prioritization 24
3.1.1 T-wise generation 24
3.1.2 Multi-objective approaches 26
3.1.3 Generating configurations from a feature model with constraint solvers 27
3.2 Configuration evaluation and mutation 28
3.2.1 Mutation analysis 28
3.2.2 Fault detection ability 28
3.3 Reverse-engineering and re-engineering software product lines 29
3.3.1 Extracting a software product line 29
3.3.2 Automated improvement and fixing of extracted software product lines 30

3.1 Configuration generation and prioritization

This section discusses work related to the generation and prioritization of configurations, which is mainly related to Part II and Part III.

3.1.1 T-wise generation

SPL t-wise testing approaches typically fall into two categories: constraint-based and search-based.

3.1.1.1 Constraint-based approaches

Since t-wise testing is difficult due to the presence of constraints, the use of constraint solving solutions have been investigated. In [CDS07], Cohen *et al.* examine the impacts of constraints and present techniques to integrate constraints handling into existing CIT tools. In Perrouin *et al.* work [PSK+10], a solution based on Alloy, a SAT solver, was devised. The approach was non-predictable in terms of generated solutions and strategies to improve scalability were proposed. Oster *et al.* [OMR10] optimized the problem upfront by flattening the FM and using CIT algorithms [CDFP97, LT98] within a dedicated constraint solver, producing predictable solutions. Both cannot handle thousand-sized FMs.

Recently, SPLCAT [JHF11], used as a reference in Chapter 4 has been proposed. SPLCAT operates by generating a *covering array* [CDS07]. In a covering array, the rows represent the configurations while the columns represent the features. The approach is incremental and adds configurations (rows) until all the feature combinations are covered. Each configuration added in the array tries to exercise the maximum number of interactions that remain to be covered. This is performed using a SAT solver which, given assumptions representing the interactions to be covered, returns a configuration. Configurations are added until all the interactions of the FM are covered. The generation technique introduced in Chapter 4 tries to maximize the dissimilarity of set of n configurations, where n is predefined. Thus, SPLCAT generate all the configurations needed to achieve 100% of t-wise coverage. On the contrary, our approach aims at maximizing the t-wise coverage of the n configurations. SPLCAT handles larger FMs than the other techniques, but it does not scale well. An improvement of SPLCAT has been recently proposed [JHF12a]: it handles larger FMs tha SPLCAT but it is limited to $t = 3$.

Logic was also used. Calvagna *et al.* explain how to deal with constraints in CIT [CG08] by encoding them in first order logic. They offer various reduction algorithms to simplify them and used a model checker to solve them. Since this work was not related to FMs, it is difficult to assess its scalability. Hervieu *et al.* [HBG11] also use reduction techniques in the aim of finding the minimal test suite in a Prolog-based implementation. However, this approach does not scale well to FMs of over 200 features, according to our experiments.

Cohen *et al.* [CDS06] proposed a relational model to represent the semantic basis for defining a family of coverage criteria for testing a SPL, such as variability coverage. CIT is then used to generate configurations that achieve a desired level of coverage. In our work, we focus on the notion of t-wise interaction coverage while generating configurations. The testing of the SPL itself is not considered. Finally, characterizing the features combinations responsible for failures can be performed

with classification trees [YCP06]. This is part of debugging and thus falls out of the scope of the present dissertation.

3.1.1.2 Search-based approaches

Search-based techniques, or more generally metaheuristic searchs have been shown to be an effective approach to solve the test cases generation problem. To this end, the test cases generation problem is reexpressed a search one. In that context, Ali *et al.* [ABHPW10] performed a systematic review on search-based approaches for testing. While underlying the limitations of these approaches, they conclude that search-based approaches are promising for coping with test cases generation problems.

Due to the computational complexity of t-wise testing of SPLs, using search-based heuristics is an option. However, we are only aware of two approaches [GCD11, EBG12]. Garvin *et al.* [GCD11] report on their experience applying and improving an extension to the AETG algorithm [CDFP97] using simulated annealing. The simulated annealing approach incrementally populates a *constrained covering array* [CDS07]. Here, each change to the value of the features is controlled by a SAT solver to ensure it is legal with respect to the FM constraints. Changes are guided by a fitness function defined over the remaining pairs to be covered: the fewer pairs to be covered, the lower the probability to make a change.

As it is shown in Section 4.3.1 of Chapter 4, using t-wise coverage as a fitness function induces scalability issues which may be intractable for very large FMs or high t values. Similarly to ours, Ensan *et al.* devised a genetic algorithm approach to generate SPL test configuration suites [EBG12]. They propose an approach where each gene is a feature to be mutated and where crossover is applied. The crossover induces possible invalid products which need to be removed and thus they face scalability issues. Their fitness function indirectly measures coverage by evaluating the variability points to be bound and the constraints concerned by the features of a configuration. On the contrary, our approach copes better with large FMs ([EBG12] does not scale over 300 features) and does not produce invalid configurations (since a configuration is always replaced as a whole). As opposed to other approaches, Ensan *et al.* and our approaches yield partial t-wise coverage due to the choice of the fitness function. This, however, allows dealing with time and cost constraints, looking for a “good enough” solution. Rubenstein *et al.* [ROZ97] proposes an algorithm that performs a search until some point in time in the context of software systems analysis. They use a measure for the accuracy of the analysis, which is also used to decide when to stop the process. This measure can be seen as the fitness function in our work. The difference with our approach is that the fitness function only evaluates the quality of the proposed solution, but does not indicate when to stop the algorithm. In our context, the stopping criteria is the time budget allowed.

Besides, the variations in space (different configurations one can form from the FM) but also in time (product versioning) have been investigated in some work [MI07, RE12]. In this work, we focus only on configuration space variations since we only consider one version of the SPL. Our approach can be adapted in the context of different version of a SPL by focusing only on the features that changed from one version to the other. Section 4.6.3 gives more details about the adaptation of our approach to SPLs evolving over time. Surveys [ER11, DMSNCMM$^+$11] report that SPL testing goes beyond the configuration generation. We do not strive to cover the full SPL testing process. Indeed, we focus on scalable configuration generation, an open research issue [AB12, GOA05, NL11]. An exact solving technique can only be used for moderate size search spaces. In this dissertation, we focus on larger SPLs where the configuration space cannot be fully explored.

Generating configurations based on mutation of the FM, as presented in Chapter 5, is the first approach to do so. It is based on a simple hill-climbing-based approach in conjunction with a SAT solver. The configuration generation process is guided by the MS. Mutation has been widely used for the purpose of testing and test generation, e.g., [HJL11, FZ12]. In our work, we used FMs represented

as Boolean formulas from which we created mutants. There are several work who investigated the mutation of logic formulas. For instance, Gargantini and Fraser devised a technique to generate tests for possible faults of Boolean expressions [GF11]. More details regarding mutation are given in Section 3.2.1.

Finally, there is work focusing on test case generation for each software product, e.g., [NFLTJ04, NTJ06, NPLTJ03, XCMR13]. In this dissertation, we only generate configurations of the software products, not the actual test cases to test them.

3.1.1.3 Prioritization

As surveyed by Nie and Leung [NL11], efforts have been made to prioritize test suites. For instance, Bryce and Colbourn [BC07] use search-based techniques (e.g., hill climbing) to select the "best test" in terms of t-wise coverage. Our goal is similar but we focus on configurations. Additionally, the proposed techniques offer improvements over the "natural" ordering provided by the AETG algorithm [CDFP97] in line with our experimentations. However, computing t-wise coverage for each configuration is expensive, especially for constrained cases, which are not taken into account in their approach and thus unsuitable in the SPL context. There are also work in the context of regression testing, e.g., [QCR08].

Yoo *et al.* [YHTS09] introduced a cluster-based prioritization technique to reduce the number of 2-wise interactions. The idea is too regroup similar test cases into clusters, and prioritize the clusters. In our work, we use a notion of similarity to compare test configurations. Prioritization of test configurations according to parameter interactions has also been done by Sampath *et al.* [SBV+08] in the context of web applications. Bryce and Memon [BM07] proposed a technique to prioritize configurations according to the t-wise interactions covered. It is a greedy approach which selects the configurations exercising the maximum number of interactions that are not already covered. The difference is that our approaches are not impacted by the t-wise interactions since they are independent of t. Indeed, our techniques select the most dissimilar configurations instead of those covering the highest number of interactions. Other work [BSPM11] also adds the notion of cost of the test to the combinatorial interaction coverage metric. In our work, we focus only on the t-wise interactions, assuming that all the configurations have the same cost. Finally, there are SPL-dedicated efforts, also in the context of test generation, but not directed to t-wise, such as Uzuncaova *et al.* [UKB10] work.

3.1.2 Multi-objective approaches

This section discusses first the related approaches that handle many objectives, then few objectives and finally the use of search approaches in software engineering.

One of the first approaches that aimed to optimize multiple configuration goals is attributed to Olaechea *et al.* [OSCR12]. This method uses a special form of FMs, called attributed FMs which record quality attributes for the features. This technique uses exact solving and consider FMs with one to three objectives. However, it fails to scale due to the computation of the exhaustive search it performs. The authors used FMs with up to 12 features. Sayyad *et al.* [SMA13] proposed the use of advanced multi-objective evolutionary algorithms for SPLs with five objectives to optimize. They experiment with five algorithms and conclude that the **indicator-based evolutionary algorithm (IBEA)** is the most suitable one for the SPL context. Sayyad *et al.* were the first, to the authors knowledge, to use constraint violation as an objective for the search process. This approach was later extended by Sayyad *et al.* [SIMA13b] with the aim of scaling to large SPLs. It is this technique, i.e.,

IBEA, that is investigated by Chapter 6. However, as shown by our results, the proposed techniques, our approaches are by far more effective.

Most of the existing approaches generate configurations that conform to the FM constraints while optimizing a single other objective. Benavides *et al.* [BTRC05] imposed constraints modeling extra-functional properties of the SPL features. They then applied constraint satisfaction solvers to generate all the possible configurations, the optimal ones etc. Another attempt to optimize the extra-functional properties of configurations was by White *et al.* [WDS09]. They proposed to transform the configuration problem into a multi-dimensional multi-choice knapsack problem to use known techniques to tackle it. White *et al.* [WGS+14] developed a tool for the multi-step configuration of evolving FMs. They show that it is possible to derive configurations automatically by mapping the SPL configuration problem to a constraint satisfaction one. Aiming at t-wise coverage, Perrouin *et al.* [PSK+10] developed a tool based on the Alloy SAT solver. Other work, e.g., [HPP+13a, HPP+13d] used a SAT solver to generate valid set of configurations. Unlike the methods presented in Part III, these approaches optimize only a single objective, i.e., either the t-wise coverage or some form of attribute coverage. Furthermore, these approaches fail to scale to large FMs.

Perhaps the closest work to the one presented in Part III is the one of Guo *et al.* [GWW+11]. Guo *et al.* propose the use of a genetic algorithm to tackle multiple objectives. To achieve this, it aggregates all objectives into one, thus using a single fitness function. This practice fails to produce a wide range of configurations and results in a single configuration that is only optimized for a specific objective weighting scheme. Additionally, it uses a repair process to make the candidate solutions valid with respect to the constraints of the FM. This process restricts the search process [SMA13]. Since it was evaluated on artificial models and thus, it is currently unclear whether it can provide satisfactory solutions for real word FMs. Our study involves the satisfaction of multiple objectives for large, heavily constrained and real-world FMs such as the Linux kernel.

Finally, regarding evolutionary algorithms, Konak *et al.* [KCS06] proposed a tutorial on the use of these kind of methods for multi-objective optimization purposes. Ensan *et al.* [EBG12] proposed a genetic algorithm approach where each gene is a feature. The crossover can thus produce invalid products. Furthermore, the explored space may contain invalid products. Their fitness function measures coverage by evaluating the variability points to be bound and the constraints concerned by the features of a product. In our approach, we use a SAT solver to only explore the space containing products valid towards the FM. The modeling of genes is performed at the product level and the crossover and mutation operators introduced avoid the introduction of invalid products.

3.1.3 Generating configurations from a feature model with constraint solvers

Approaches for the automated analysis of FMs have proliferated these last 20 years [BSRC10]. Such techniques enable to extract information from the FM, such as identifying the mandatory features or count the valid configurations of a SPL. These techniques rely on binary decision diagrams or solvers, such as SAT or **satisfiability modulo theory (SMT)** solvers [DMB11]. The efficiency of these techniques has been investigated by Pohl *et al.* [PLP11] with the conclusion that these approaches induce a certain overhead and that there is still room for improvement. The use of SAT solvers for reasoning on FMs has been reported as being an easy task [MWC09]. In this work the authors conclude that the previous reports on the efficiency of SAT solvers is not incidental in practice. The FMs used in this work are extracted from existing code such as the Linux kernel. To the best of our knowledge, the efficiency of automated analysis techniques have not been investigated on such models. Finally, augmenting the features of FMs with quality attributes such as cost, as performed in Chapter 7 has been used in several previous studies, e.g., [SMA13, SIMA13b, OSCR12, HPP+13c, ZYL11].

3.2 Configuration evaluation and mutation

In this section, existing techniques used for evaluating configuration along with mutation analysis are discussed.

3.2.1 Mutation analysis

Mutation analysis is a powerful technique with multiple applications [Off11, JH11]. Generally, code-based mutants have been used to guide the test generation process [PM11, PM12, FZ12], to assist the debugging activities [PLT13, PLT12] and to evaluate the fault detection ability of a test suite [GGZ+13, ABLN06]. The technique has also been applied to test specification models [JH11] and to capture semantic errors of the programs [CDH13]. For instance, Mottu *et al.* used mutation to test model transformations [MBLT06]. Other applications of this technique include Petri nets [FMM+96] or security policies metamodels [MFB08].

Contrary to the above-mentioned work, the present dissertation applies mutation analysis to the FM as performed in Chapter 8 or to a model of the program inputs, then measuring the correlation between model-based mutants and code-based faults (Chapter 9). The Kendall coefficient used in Chapter 9 has been used in several work to measure the the correlation between two measured quantities. For instance, Gligoric *et al.* [GGZ+13] performed a correlation analysis using this coefficient in order to evaluate the relationship between coverages and MS. In this work, we perform a correlation analysis by measuring the Kendall τ between a) the number of input parameter interactions covered by a given test suite and b) the number of the introduced mutants distinguished by the test suite with its actual fault detection.

Considering Boolean specifications, mutation faults have been used to select minimum test suites [GF11]. Similarly, Kaminski *et al.* [KPAO11] use a logic mutation approach to measure test data quality. Their approach relies on the notion of higher order mutants [JH09] and aim at improving logic-based testing. In another work, Kaminski *et al.* [KAO13] target at augmenting logic-based criteria inspired by the mutation approach. Contrary to these approaches, we apply mutation on the logic underlying these models. Andrews *et al.* showed that generated mutants can be used to predict the detection effectiveness of real faults [ABLN06]. They investigate the relative cost and effectiveness of different testing coverage criteria. Contrary to Just *et al.* [JJI+14], we do not focus on whether or not the generated mutants of the model are representative of real defects.

3.2.2 Fault detection ability

Most of the work on SPL testing was focused on providing scalable and efficient test generation techniques but less attention has been devoted to the evaluation of the bug detection ability of generated test suites, motivating this research. In [SOLF12], Steffens *et al.* provide an industrial account on the actual detection ability of t-wise techniques, showing that they actually detect bugs. Johansen *et al.* [JHF+12c] applied such techniques on the Eclipse development environment and exhibited some interaction problems. Both did not consider issues occurring in the FM itself. Ensan *et al.* [EBG12] developed an fault injection tool which associate errors to construct of the FM such as individual features, groups or constraints. This fault injection tool aims at simulating actual issues

found in practice. To the best of our knowledge, our approach presented in Chapter 8 is the first to evaluate the ability of dissimilar test suites to detect FMs errors.

CIT is a well researched technique with multiple criteria and combination strategies [GOA05]. However, very few work consider the fault detection ability of CIT, e.g., [PYCH13, KWG04, CDS08, GLOA06, SNX05, BM07]. In the most recent one, Petke *et al.* [PYCH13] show that higher t strengths result in finding more faults than lower strengths. Our work differs from this one in three ways. First, we use mutation while they only consider t-wise. Second, we consider the correlation between faults and t-wise interactions. They only investigate whether covering higher interaction strengths results in higher fault detection. Third, we use randomly selected test suites while they use test suites selected with a covering array tool [CCL03]. Similarly, Arcuri and Briand [AB12] showed that random testing can perform similarly to CIT on large-scale models. However, their results hold only in the case where there are no input constraints.

3.3 Reverse-engineering and re-engineering software product lines

This section discusses work related to our contributions presented in Part V.

3.3.1 Extracting a software product line

Whether extracting a product line is useful or not has been assessed in [BRR10]. In this work, Berger *et al.* investigated the assessment of SPVs to extract a product line. They propose a set of metrics that enable the software architects and project managers to estimate whether it is beneficial or not to construct a product line. This approach is complementary to our and can be done as a prior step to our approach.

There are few work related to the extraction of a SPL from the source code of SPVs. Yoshimura *et al.* [YNHK08] propose an algorithm to detect variability across the source code of a collection of existing products. This method only extracts *factors* to specify the variability. In [KK12], Klatt *et al.* propose ay reverse-engineering process for variability. This work also abstracts input SPVs using abstract syntax tree models. The extraction of the SPL implementation is not considered in these studies.

Zhang *et al.* [ZB13] present a framework for re-engineering variability. However, this work only focus on the extraction of variability from the source code with conditional compilation directives. Xi *et al.* [XXJ12] propose an approach based on formal analysis concept for the identification of code units that are associated with a set of existing features. Indeed, in addition to the source code of SPVs, this approach also takes as input data the list of features associated to each product variant. It then tries to locate the code units associated to each feature. The difference between this work and our approach presented in Chapter 10 is that *ExtractorPL* only considers the source code of the SPVs as input data, without any additional information.

To the best of our knowledge, there is no work related to the extraction of a full implementation of a SPL from the source code of SPVs, i.e. a SPL which allows to generate and compose the code of the extracted features as long as proposing a FM. There are some existing extractive approaches that only consider the feature identification step, e.g. [SLB+11]. In this case, a FM with constraints is extracted. In our approach, we build a FM without constraints, but we propose the full implementation of the SPL.

Extracting the variability from other assets than source code has been investigated in several work, e.g., [DDH+13]. In [RC12], Rubin *et al.* propose an approach to extract a product line from architectural artifacts. Frenzel *et al.* [FKBA07] use the reflexion method to refactor a collection of product architectures into a SPL architecture. In their work, variability is specified using annotations. In [ACC+14], FMs are extracted from plugin dependencies and architecture fragments of SPVs. Yssel *et al.* [RPK10] consider the extraction of FMs from a set of similar models that represent function blocks, a kind of architectural models for embedded systems. *ExtractorPL* can be modified to use architectural artifacts. Indeed, feature structure trees can be employed to abstract architectural models [AKL09].

Finally, while *ExtractorPL* extracts a SPL from the source code of a set SPVs, Valente *et al.* [VBP12] propose a semi-automatic approach where a SPL is extracted from a single software product.

3.3.2 Automated improvement and fixing of extracted software product lines

Efforts have been made to reverse-engineer FMs. She *et al.* introduced procedures [SLB+11] to recover constructs such as mandatory features or implies edges to build graphs and provided ranking heuristics to allow the modeler identifying the features hierarchy. In the same context, an approach that builds FMs from the feature sets describing the system variants based on Evolutionary Algorithms has been proposed [LHGB+12]. Acher *et al.* [ACC+11] focused on maintaining a link with the SPs while reverse engineering FMs. They also took into account the architect's knowledge to build FMs consistent with the software architecture. The method presented in Chapter 11 of this dissertation complements these approaches by introducing a way of validating the FM they provide. In addition, our technique goes a step further by automatically fixing the inconsistencies it identifies.

To debug configurations, Hubaux *et al.* [XHSC12] proposed a fix-generation approach, called *range fix* to prevent wrong configurations. Their strategy uses constraints solving to propose a list of valid assignments for enabling features. This is done using an underlying model which contains the system constraints. In [TSD+12], Tartler *et al.* presented an approach to find and fix defects contained in implementations of configurations of large-scale systems. It is a diagnostic tool which aims at fixing the code. In the same lines, Segura *et al.* [SGB+12] presented a framework for benchmarking and testing on the analysis of FMs. This approach is able to automatically detect faults in the analysis tools that operates on a FM. Our approach operates on the FM itself and aims at testing whether its constraints are consistent, and if they are not, to fix them.

Search-based techniques have also been used to perform automatic program improvement. In [LH12], Langdon *et al.* proposed an approach to improve the lines of code of a system. Based on a fitness function and population of patches, they evolved the original code into a faster version while keeping the semantic of the code at least unchanged or better. Similarly, in [LGDVFW12], Le Goues *et al.* presented a scalable genetic cloud computing oriented technique to repair erroneous programs.

Part II

Mono-objective configuration generation

4

Scalable t-wise generation and prioritization of software product line configurations

This chapter introduces scalable techniques for generating and prioritizing configurations for combinatorial testing. This is essential since existing approaches for t-wise testing fail at scaling to large software product lines.

This chapter is based on the work that has been published in the two following papers:

- C Henard, M Papadakis, G Perrouin, J Klein, P Heymans, and Y Le Traon. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Trans. Software Eng.*, 40(7):650–670, 2014
- Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Pledge: A product line editor and test generation tool. In *Proceedings of the 17th International Software Product Line Conference Co-located Workshops*, SPLC '13 Workshops, pages 126–129, New York, NY, USA, 2013. ACM

Contents

4.1 Introduction **34**
4.2 The similarity heuristic **35**
4.3 Configuration generation **36**
4.3.1 A similarity-based fitness function 36
4.3.2 A search-based approach 37
4.4 Configuration prioritization **38**
4.4.1 Local Maximum Distance prioritization 38
4.4.2 Global Maximum Distance prioritization 39
4.5 Empirical study **39**
4.5.1 Comparison with state of the art tools (research question 1) 41
4.5.2 Configuration generation assessment (research question 2) 43
4.5.3 Configuration prioritization assessment (research questions 3 & 4) 48
4.6 Discussion **52**
4.6.1 Detecting t-wise interaction faults 52
4.6.2 Practical implications 54
4.6.3 Further applications 55
4.6.4 Limitations 55
4.6.5 Threats to validity 56
4.7 Conclusions **56**

4.1 Introduction

CIT is one of the most famous criterion for generating configurations for SPLs. However, it is difficult to apply on large and heavily constrained SPLs.

The scalability issues of combinatorial interaction testing. Computing all the t-wise interactions in the presence of constraints, as it is the case for FMs, is a hard problem to solve [JHF11, PSK$^+$10]. Although t-wise generation techniques from FMs have been greatly improved over the last years, they still face scalability issues in the presence of constraints [AB12, GOA05, NL11]. Therefore, dealing with large FMs such as those used in industry is still an open research issue. Furthermore, the CIT literature points out the need for dealing with higher interaction strengths ($t > 2$) [KLK08, RSM$^+$10, PYCH13]. Preliminary evidence shows that 3-wise interactions may commonly appear in SPL testing practice [SOLF12] and that higher interaction strengths are important in achieving a higher fault detection [PYCH13]. Additionally, the results of the present study show that state of the art CIT tools fail to scale even on FMs of moderate size for high interaction strengths ($t = 3, 4$). Since such strengths may remain out of reach, one may ask if it is possible to cope with these difficult situations by relaxing the t-wise criterion. This leads us to the question of whether *we can mimic t-wise configurations generation, partially but efficiently while achieving decent coverage?*

The need for prioritization. While t-wise testing drastically reduces the number of configurations to consider, this number may still be too high to fit the budget allocated for SPL testing. For example, 2-wise coverage for the Linux FM (over 6,000 features) already requires 480 configurations to be tested [JHF12a]. This observation is in line with Song *et al.* 's [SPF12] motivation : since they reported that key interactions may involve up to 7 option settings, full 7-wise coverage of a realistic system (admitting such a computation is possible) may yield too many configurations to consider. Therefore, being able to prioritize configurations is critical from a practical point of view.

Contributions of this chapter. The work presented in this chapter is motivated by the results of Arcuri and Briand [AB12] who showed that, in the case of large models, random testing is competitive with CIT for finding interaction faults. Hence, they suggest the use of random testing as a possible way to circumvent the scalability issues of the CIT approaches. Unfortunately, real world applications involve constraints between features and the results of Arcuri and Briand do not hold when constraints are present.

We propose two approaches working with constraints capable of generating and prioritizing configurations for CIT: (a) a randomized approach, named here as unpredictable, and (b) a SB technique. Following the lines of Arcuri and Briand, we empirically investigate the probability of finding an interaction failure for $t = 2, ..., 6$. Our SB approach efficiently generates valid configurations, i.e., respecting the constraints of the FM, for t-wise testing. The innovative part of the proposed approach is that it is independent of t and able to operate on large and constrained models by both generating and prioritizing configurations. The applicability of the proposed strategies is evaluated on both real and generated FMs. Our approach stands up against the comparison with existing tools for small and moderate FMs, while it is able to scale well to models with $6,000$ features for t up to 6. In summary, the present chapter of this dissertation provides the following insights:

- We show that state of the art tools for CIT face severe scalability issues with large SPLs.
- We propose a randomized and a SB approach able to generate valid configurations for t-wise testing for large SPLs.

- We introduce two scalable configuration prioritization techniques.
- We perform a wide empirical study including FMs that contain more than 6,000 features.

The remainder of this chapter is organized as follows: Section 4.2 introduces the heuristic used by the approaches. Sections 4.3 and 4.4 respectively detail the configuration generation and prioritization techniques. Section 4.5 reports on the empirical study. Finally, Section 4.6 discusses the proposed approaches before Section 4.7 concludes the chapter.

4.2 The similarity heuristic

Similarity is a heuristic used here to compare two configurations. In model-based testing, it has been found that dissimilar test suites have a higher fault detection power than similar ones [HB10]. The results presented in this chapter (see Section 4.5) suggest that two dissimilar configurations are more likely to cover a greater number of valid t-sets than two similar ones.

In this context, we define a distance measure d between two configurations to evaluate their degree of similarity. Since a configuration is considered as a set of selected or unselected features. Thus, a straightforward distance measure is a set-based one, like the Jaccard distance [Jac01] or any other set-based distance metrics such as the Dice or Anti Dice measures [HB10]. If C represents all the possible configurations of a FM, the Jaccard distance is mathematically given by:

$$d: \begin{array}{rcl} C \times C & \longrightarrow & [0, 1.0] \\ (C_i, C_j) & \longmapsto & 1 - \dfrac{\# C_i \cap C_j}{\# C_i \cup C_j}. \end{array}$$

The resulting distance varies between 0 and 1. More particularly, a distance equal to 1 indicates that the two considered configurations are completely different. A distance equal to 0 denotes that the two configurations are the same (redundant). It is noted that an unselected feature is also an element of the set representing a configuration.

For instance, consider the following configurations of the FM of Figure 2.1:

$$\begin{array}{l} C_1 = \{+f_1, +f_2, +f_3, +f_4, +f_5, -f_6, +f_7, -f_8, +f_9\}, \\ C_2 = \{+f_1, +f_2, +f_3, +f_4, +f_5, +f_6, -f_7, -f_8, +f_9\}, \\ C_3 = \{+f_1, +f_2, +f_3, +f_4, +f_5, +f_6, +f_7, -f_8, +f_9\}, \\ C_4 = \{+f_1, +f_2, +f_3, -f_4, +f_5, +f_6, -f_7, +f_8, -f_9\}. \end{array}$$

With these configurations, $d(C_1, C_2) = 1 - \dfrac{\#\{+f_1, +f_2, +f_3, +f_4, +f_5, -f_8, +f_9\}}{\#\{+f_1, +f_2, +f_3, +f_4, +f_5, -f_6, +f_6, -f_7, +f_7, -f_8, +f_9\}} = 1 - \dfrac{7}{11} \approx 0.36$ and $d(C_1, C_4) \approx 0.71$. In this example, C_1 and C_2 are the most similar configurations (they share the lowest distance), whereas C_1 and C_4 are the most dissimilar ones. Thus, if we had to choose only two configurations, C_1 and C_4 would be the most likely to cover the greatest number of t-sets according to the similarity heuristic.

4.3 Configuration generation

In this section, we take benefit from the similarity heuristic to guide the generation of configurations. In this context, we want to maximize the t-wise coverage achieved by the configuration suite CS with a time t allowed for generating the configurations. Toward this direction, we introduce an approach, based on the (1+1) Evolutionary Algorithm [DJW02]. Specifically, the configuration generation problem is formulated as a SB one. The space of all the valid configurations is defined as the search space. Thus, meta-heuristic techniques can be used in order to efficiently explore this space. In view of this, similarity is used as a fitness function towards searching for configurations in this space. It enables: (a) a computationally interesting approach, independent of t and (b) prioritizing the generated configurations without necessitating much additional computation.

4.3.1 A similarity-based fitness function

Our intuition, which will be confirmed in Section 4.5, is that the similarity heuristic is a relevant choice to define a fitness function f to evaluate a configuration suites. Thus, if we consider a configuration suite of m configurations $CS = C_1, ..., C_m$, f is formally defined as follows:

$$f : \begin{array}{ccc} C^m & \longrightarrow & \mathbb{R}_+ \\ C_1, ..., C_m & \longmapsto & \sum_{j>i}^{m} d(C_i, C_j). \end{array}$$

For instance, with reference to Section 4.2, $f(C_1, C_3, C_4) = d(C_1, C_3) + d(C_1, C_4) + d(C_3, C_4) \approx 1.53$. This function, which generalizes the similarity distances for m configurations, allows evaluating the quality of a configuration suite in terms of t-wise coverage. Indeed, the information conveyed by this function is: the higher the fitness value of the given configuration suite of m configurations, the higher the distances between the configurations, resulting in a potentially higher t-wise coverage.

Although evaluating the exact coverage would be a natural choice for a fitness function, say f_c, it would be computationally expensive for such a use. Indeed, for each configuration, it requires computing all the t-sets covered by this configuration. Consider a FM with n features and m configurations. If $\binom{n}{k}$ denotes the binomial coefficient, f_c requires to compute:

$$N = m\binom{n}{t} = \frac{mn!}{t!(n-t)!} \tag{4.1}$$

t-sets to evaluate the coverage of the whole configuration suite, which represents N operations. On the contrary, f requires $N' = \binom{m}{2} = \dfrac{m(m-1)}{2}$ distances computation plus the sum evaluation, which represents m additions.

We assume that $2 \leq t \ll n$. Therefore, the time required to compute one particular distance between two given configurations is small compared to the coverage evaluation of these two configurations, i.e., $N \gg N'$. Indeed, f does not depend on t. We also assume that one will test fewer configurations than the number of features, and thus that $m \ll n$. Especially, in a realistic and industrial context (with large FMs), the testing process is usually subjected to time and budget limitations. It thus does not allow testing as many configurations as features. It results that $N \gg N'$ and even more while t increases. Recall that we focus on t-wise, for high t-values. This fact implies a computationally lower cost for f compared to f_c. As a result, f is used as the fitness function for the configuration generation.

4.3.2 A search-based approach

Classical constraint-based t-wise techniques, e.g., [JHF12a], are unable to scale to large FMs and to high values of t. This is mainly due to the number of t feature combinations. The proposed approach, which is independent of t, is composed of two steps. The first one is the generation of valid configurations using a SAT solver, and the second one is the configuration selection. The search process is formed by iteratively repeating these two steps. A similar technique that combines constraint solving and SB approaches in a scalable way has been proposed by Harman *et al.* [HJL11] for mutation-based test generation.

4.3.2.1 Generating configurations

A SAT solver is used to produce valid configurations. Once a FM is converted into a Boolean formula [MWC09], the solver can generate valid configurations. As a result, a search space containing only valid configurations is formed.

Typically, a configuration is a satisfiable "model" for a given SAT solver [LBP10]. To this end, the literals of the logical clauses (i.e., clauses represent the constraints of the FM) are assigned values. If the constraints are satisfied, one configuration is returned. However, assignments to the literals are done in a particular order which involves the following problem: no uniform exploration of the space of all the valid configurations is possible. Indeed, the order used by the solver to parse the logical clauses and literals enables their prediction. In that case, the approach always returns the same solution in a deterministic way. As a result, the configurations enumeration is driven by the order used by the solver.

To overcome this issue, and thus to get configurations in an *unpredictable* way, one solution is to randomize how the solver parses the logical clauses and the literals and how it assigns values to variables. It thus makes the solving process entirely randomized. It prevents predicting the next configuration that will be returned. Additionally, it allows selecting configurations from the full space instead of enumerating them in a predictable order. Since it enables the use of SB approaches in a non-biased way, the unpredictable strategy forms a contribution of this chapter.

4.3.2.2 Selecting configurations

The objective is to generate a configuration suite CS of m configurations. The proposed approach is formalized in Algorithm 1. Informally, the SB method starts by selecting m configurations in an unpredictable way (lines 5 to 8). Then, these configurations are evaluated by the fitness function f (line 11) and prioritized (line 12). The technique used to prioritize the configurations (line 12) is presented in Section 4.4.2. These configurations define the initial list CS. Then, by using the distances computed while evaluating f, the worst configuration is determined. The worst configuration is the one which has the lowest participation in the fitness function. In other words, it is the last element of CS (line 13). The next step consists of trying to replace this configuration by an unpredictable one got from the solver (lines 14 to 16). This replacement is conserved if and only if the fitness of the resulting list increases (lines 17 to 20). This whole process is repeated during a certain allowed amount of time t.

This technique can be considered as a genetic algorithm without crossover. It is thus an adaptation of the (1+1) Evolutionary Algorithm [DJW02]. Indeed, instead of removing a random configuration, the worst ranked configuration, in terms of fitness, is removed.

Combining constraints with SB methods forms a suitable approach for the configuration generation. Its use differs from both SB [ABHPW10] and similarity-based techniques [HB10]. Indeed, without constraint solving, generating valid configurations is almost impossible for large scale FMs. This problem is shortened by combining similarity, constraint solving and SB approaches.

4.4 Configuration prioritization

In this section, the similarity distances are used for prioritizing a given configuration suites, no matter the way they have been obtained. To this end, two algorithms named *Local Maximum Distance* and *Global Maximum Distance* are introduced. They produce a list CS, which is the result of the prioritization. They enable prioritizing efficiently the configurations with respect to t-wise.

4.4.1 Local Maximum Distance prioritization

Algorithm 2 formalizes this procedure. This approach iterates over the initial unordered configuration suites S, looking for the two configurations sharing the maximum distance (line 6). These two configurations are then added to the resulting list CS and removed from S (lines 7 to 9). This process is repeated until all the configurations from S are added to CS.

Algorithm 1 Search-based configuration generation(m, t)

```
1: input: m, t          ▷ Number of configurations to generate and execution time allowed for generating them
2: output: CS           ▷ configuration suite (prioritized)
3: CS ← []
4: S ← ∅                ▷ Set of configurations
5: for i ← 1 to m do
6:     C_unpredictable ← Request to the solver      ▷ Reinitialize the solver if it cannot give a new configuration
7:     S ← S ∪ {C_unpredictable}
8: end for
9: s ← size(CS)
10: while the elapsed time is lower than t do
11:     fitness ← f(CS[1], ..., CS[s])
12:     CS ← Global Max. Dist. Prioritization(S)
13:     C_worst ← CS[s]      ▷ C_worst verifies min (Σ_{k=1}^{s} d(C_worst, CS[k]))
14:     repeat
15:         C_unpredictable ← Request to the solver  ▷ Reinitialize the solver if it cannot give a new configuration
16:     until C_unpredictable ≠ C_worst
17:     CS.set(s, C_unpredictable)      ▷ The worst configuration is replaced
18:     newFitness ← f(CS[1], ..., CS[s])
19:     if newFitness ≤ fitness then
20:         CS.set(s, C_worst)          ▷ The worst configuration is taken back
21:     end if
22: end while
23: return CS
```

Algorithm 2 Local Maximum Distance prioritization (S)

```
1: input: S = {C_1, ..., C_m}                        ▷ Unordered configuration suites
2: output: CS                                        ▷ Prioritized configuration suite
3: CS ← []
4: while #S > 0 do
5:     if #S > 1 then
6:         Select C_i, C_j ∈ S where max(d(C_i, C_j))   ▷ Take the first one in case of equality
7:         CS.add(C_i)
8:         CS.add(C_j)
9:         S ← S \ {C_i, C_j}
10:    else                                          ▷ S contains only one element
11:        CS.add(C_i) where C_i ∈ S
12:        S ← ∅
13:    end if
14: end while
15: return CS
```

Algorithm 3 Global Maximum Distance prioritization (S)

```
1: input: S = {C_1, ..., C_m}                        ▷ Unordered configuration suites
2: output: CS                                        ▷ Prioritized configuration suite
3: CS ← []
4: Select C_i, C_j ∈ S where max(d(C_i, C_j))        ▷ Take the first ones in case of equality
5: CS.add(C_i)
6: CS.add(C_j)
7: S ← S \ {C_i, C_j}
8: while #S > 0 do
9:     s ← size(CS)
10:    Select C_i ∈ S where max(Σ_{j=1}^{s} d(C_i, CS[j]))   ▷ Take the first one in case of equality
11:    CS.add(C_i)
12:    S ← S \ {C_i}
13: end while
14: return CS
```

4.4.2 Global Maximum Distance prioritization

This approach is formally described in Algorithm 3. Informally, this approach selects at each step the configuration which is the most distant to all the configurations already selected during the previous steps. To this end, the two configurations belonging to S and sharing the highest distance are first added to CS (lines 4 to 6). These two configurations are then removed from S (line 7). The next step consists in adding to CS and removing from S the configuration sharing the maximum distance to all the configurations already added to CS (lines 8 to 13): for each configuration of S, we sum the individual distances with the other configurations of CS, thus giving a value for the set. Then the maximum is obtained by comparing these set values (line 10). This process is repeated until S is empty.

This technique allows having more diversity than the *Local Maximum Distance* one for $k < m$ configurations, but it is computationally more expensive. This is due to the need of calculating all the distances from one configuration to the others (Alg. 3, line 10).

4.5 Empirical study

In this section, the configuration generation and prioritization approaches are assessed. In configuration generation, we aim at selecting configurations providing the highest coverage. In configuration

Table 4.1: The 110 moderate size feature models involved in the empirical study.

	10 real FMs [MBC09]										100 generated FMs				
	Cellphone	Counter Strike Simple FM	SPL SimulES, PnP	DS Sample	Electronic Drum	Smart Home v2.2	Video Player	Model Transformation	Coche Ecologico	Printers	20 FMs	20 FMs	20 FMs	20 FMs	20 FMs
#Features	11	24	32	41	52	60	71	88	94	172	15	50	100	200	500
#Constr.	22	35	54	201	119	82	99	151	191	310	31.65	94.7	195.6	395.7	983.2
#Config.	14	18,176	73,728	6,912	331,776	3.9E9	4.5E13	1.7E13	2.3E7	1.1E27	209.55	1.0E8	8.6E15	3.2E20	8.4E80
#2-sets	151	833	1,448	2,592	3,746	6,189	7,528	13,139	11,075	42,638	300.65	4,103	17,368	71,760	4.67E5

prioritization, the emphasis is on maximizing the overall t-wise coverage each time a configuration is tested. The objective of this case study is to answer the four following research questions (RQs):

- [RQ1] *How does our configuration generation approach compare with state of the art tools?*
- [RQ2] *How effective is the configuration generation approach when applied on both moderate and large size FMs?*
- [RQ3] *How do our prioritization approaches compare with an interaction-based technique?*
- [RQ4] *How effective are the configuration prioritization approaches when applied on both moderate and large size FMs?*

Answering the first RQ amounts to evaluating how our configuration generation approach performs compared to state of the art techniques. We expect our approach to provide a t-wise coverage close to the one achieved by the examined tools. The second RQ aims at evaluating the configuration generation approach on both moderate and large FMs. Unlike existing tools, we expect our approach to scale up to $t = 6$ even on large FMs, by providing a partial but scalable t-wise coverage. We also expect it to provide higher coverage than a random technique for selecting the configurations. The third RQs amounts to evaluating how our configuration prioritization approaches perform compared to a state of the art technique based on interaction coverage. We expect our approach to provide a t-wise coverage close to the state of the art with a considerably lower execution time required as it bypasses the t-wise computation. Finally, the fourth RQ aims at evaluating how our two prioritization techniques perform on both moderate and large FMs, by comparing them with a random approach. We expect our prioritization approaches to perform better than a random one.

Empirical results regarding the stated RQs are presented and analyzed. The conducted experiments[i] are performed on a Quad Core@2.40 GHz with 24GB of RAM. The study employs 114 FMs[ii] divided into two categories. The first 110 FMs are small to medium size (with a number of features lower or equal to 1000); they are referred to as the *moderate* size FMs. A second subset is composed of 4 FMs of large size; they are referred to as the *large* FMs.

Regarding the moderate size FMs, 10 of them are real and 100 are artificially generated. The real FMs are taken from [SLB+11, MBC09] while the artificial ones are produced with the SPL Online Tools (SPLOT) FM generator [MBC09]. All involved FMs are consistent (i.e., the constraints are possible to fulfill). Details about the moderate FMs are recorded in Table 4.1. For each FM, it presents the number of features, the number of valid configurations[iii] and the number of valid 2-sets.

Regarding the large FMs, three are real, taken from [SLB+11] and one is artificially created. The details of these FMs are recorded in Table 4.2. It presents, for each FM, the number of features and

[i] The implemented approaches and the experimental data are available at `http://research.henard.net/SPL/`.
[ii] Handled via the SPLAR [MBC09] library and the Sat4j [LBP10] MBC09b.
[iii] Computed via a Binary Decision Diagram.

Table 4.2: The 4 large size feature models involved in the empirical study.

	eCos 3.0 i386pc [SLB+11]	FreeBSD kernel 8.0.0 [SLB+11]	Generated FM	Linux kernel 2.6.28.6 [SLB+11]
#Features	1,244	1,396	5,000	6,888
#Constr.	3,146	62,183	9,419	343,944
#2-sets	2,910,229	3,765,597	49,080,075	92,540,449
#3-sets (≈)	2.25E9	3.44E9	1.61E11	4.19E11
#4-sets (≈)	1.27E12	2.34E12	3.97E14	1.50E15
#5-sets (≈)	5.79E14	1.26E15	7.70E17	3.85E18
#6-sets (≈)	2.22E17	5.76E17	1.26E21	8.71E21

the number of valid t-sets. The number of configurations cannot be computed in a reasonable amount of time (in days) due to the high number of constraints and features of these FMs.

For the needs of the experiment, the t-sets of the FMs for $t \geq 3$ are computed using the following procedure. First, a list of all the features of the FM is recovered. Then, all the possible t-sets are enumerated and provided to the solver to determine whether they are valid or not. For the large FMs, computing the exact number of valid t-sets is a non-trivial and time consuming task. For instance, it took around 3 days to a 10-threaded program running on our system to compute the 92,540,449 valid 2-sets of the Linux FM. As t increases, the number of valid t-sets explodes. As a result, we estimate the number of t-sets. To this end, 1,000 t-wise sets are randomly sampled and checked. Since the total number of possible t-sets of a FM is known and equal to $\binom{2n}{t}$ for n features ($2n$ because each feature is either selected or unselected), the valid t-sets can be directly estimated (law of large numbers). For example, if 800 t-sets out of 1,000 sampled are valid, the number of estimated valid t-sets is equal to $\frac{800 * \binom{2n}{t}}{1,000}$.

4.5.1 Comparison with state of the art tools (research question 1)

In this section, we compare our approach with three state of the art tools: ACTS [BYL+12], CASA [CCL03] and SPLCAT [JHF12a]. The latter is the most recent covering array tool available and performs for $t = 2$ and $t = 3$. The two others perform for $t = 2$ to 6.

4.5.1.1 Experiment setup

We compare our generation approach with the three tools. We also consider CASA where the desired number of configurations can be specified. This approach is denoted as CASA-n. We employ the 10 real FMs of moderate size. For each FM and for $t = 2, ..., 6$, the three approaches are executed. Then, if the result is available, our approach and CASA-n are performed with the parameters corresponding to the minimum number of configurations provided by the other techniques. Similarly, the running time of our approach was set to the minimum one. CASA, CASA-n and our SB approach are performed 10 times independently as they are heuristic techniques providing different solutions at each execution.

Table 4.3: Comparison of the configurations' generation with ACTS, CASA and SPLCAT on the 10 smallest feature models of the empirical study for $t = 2, ..., 6$. Our search-based approach has been performed using the minimum number of configurations and minimum time performed by the other approaches, indicated in bold. N/A indicates that the generation time exceeded 3 days. The t values for which there is no result available are not represented.

		ACTS (IPOG)		CASA (avg 10 runs)		CASA-n (avg 10 runs)		SPLCAT		SB (avg 10 runs)		
FM	t-wise	Configs.	Time	Configs.	Time	Configs.	Time	Configs.	Time	Configs.	Time	Cov.
Cellphone	2	9	2.1	**7**	0.55	7	**0.05**	8	0.15	7	0.05	98.37%
	3	13	4.4	14	1.14	14	**0.045**	**13**	0.24	13	0.045	98.12%
	4	**14**	16	14	3.79	14	**0.34**	N/A	N/A	14	0.34	99.98%
	5	14	57	14	55.15	**14**	**0.27**	N/A	N/A	14	0.27	100%
	6	**14**	399	14	6,947	14	**0.45**	N/A	N/A	14	0.45	100%
C. Strike Simple FM	2	13	3.3	8.66	1,28	**8**	0.27	10	**0.24**	8	0.24	99.34%
	3	33	70	25.33	22,16	**25**	2.21	38	**0.8**	25	0.8	99.71%
	4	94	3,278	72.67	790	**72**	**24,7**	N/A	N/A	72	24.7	98.77%
SPL SimulES, PnP	2	11	7	**9**	3.67	9	0.6	10	**0.3**	9	0.3	99.41%
	3	32	211	26.33	122.9	**26**	6.49	35	**1**	26	1	99.32%
	4	83	53,602	**72**	3,026	72	**282.5**	N/A	N/A	72	282.5	99.64%
DS Sample	2	103	506	96.33	96.16	**96**	3,13	97	**0.9**	96	0.9	98.49%
	3	N/A	N/A	**385**	12,093	385	113.2	419	**4.8**	385	4.8	99.51%
Electronic Drum	2	35	38.4	23.67	4,826	**23**	4.04	27	**0.6**	23	0.6	99.46%
	3	178	43,416	N/A	N/A	134	91.22	**134**	**2.9**	134	2.9	99.91%
Smart Home v2.2	2	17	15.5	15	28	15	3.1	**15**	**0.5**	15	0.5	99.44%
	3	75	3,731	55.67	5,182	**55**	106.7	64	**3.2**	55	3.2	99.80%
Video Player	2	15	26.5	9.33	56.4	**9**	1.5	18	**0.7**	9	0.7	99.75%
	3	46	32,687	35.67	2,542	**35**	47.02	47	**3.7**	35	3.7	99.98%
Model Transformation	2	35	187	26.33	3,165	**26**	13.6	28	**0.9**	26	0.9	99.45%
	3	N/A	N/A	N/A	N/A	130	482.7	**130**	**10**	130	10	99.91%
Coche Ecologico	2	97	2,348	**90**	156.91	90	10.6	95	**1.1**	90	1.1	99.67%
	3	N/A	N/A	N/A	N/A	378	3,694	**378**	**13**	378	13	99.87%
Printers	2	186	148,446	180.8	718.82	**180**	71.8	182	**2.8**	180	2.8	99.75%
	3	N/A	N/A	N/A	N/A	560	N/A	**560**	**139**	560	139	99.81%

4.5.1.2 Experiment results

The results of the comparison are recorded in Table 4.3. When the time required by an approach exceed three days (259,200 seconds), we consider its results as not available (N/A). Along the same lines, when none of the three tools can perform for a specific t value in less than three days, the result for this t value is not presented. Following the results of Table 4.3, it is clear that SPLCAT is much faster than the two other tools on almost all the FMs. In some cases, it is more than 10,000 times faster. Only CASA-n can compete in terms of time on some FMs such as the Cellphone one. As a result, our approach has been most of the time executed with the time used by SPLCAT or CASA-n.

Regarding the size of generated configuration suites, the smaller ones are shared between CASA and SPLCAT. Our approach and CASA-n have been performed using the smallest size. Regarding the coverage achieved by our approach, one can see that it is close to 100% for all the subjected FMs.

Finally, one can observe that as the complexity of the FM increases, the tools require more time to generate the configurations and thus do not scale well to large FMs. In addition, none of the tools were able to perform for t values above 3 on most of the employed FMs, even though the complexity of these FMs is quite low.

4.5.1.3 Research question 1 summary

The experiments conducted for the comparison with state of the art tools bring out the following conclusions. First, **our configuration generation approach can compete with existing ones**. Indeed, our approach provides a partial t-wise coverage, with values very close to 100% for all the FMs studied. This was achieved using the minimum amount of time required among the three tools. Second, **existing tools have difficulties to scale to t values greater than 3, even on relatively small FMs.** Indeed, our experiments performed on moderate size FMs demonstrated that for 7 FMs out of 10, none of the three tools was able to provide results within 3 days from $t = 4$. The 3 FMs on which the tools worked are the smallest one, and for two of them, results are only available up to $t = 4$. Overall, the fastest tool among the three is SPLCAT.

4.5.2 Configuration generation assessment (research question 2)

Here, we assess the ability of the proposed approaches to cover t-sets. To this end, we evaluate our approach on all the moderate size FMs for 2-wise and compare it to SPLCAT. We limit to 2-wise since SPLCAT does not scale well to 3-wise or above (at least in a reasonable amount of time, in days) for the subjected FMs. As far as we know, our SB approach is the only one which allows scaling to any t value, even for large FMs. Finally, since no other technique can serve as a basis for comparison for the large FMs, we compare the SB approach with configurations selected in an unpredictable way from the SAT solver (Section 4.3.2.1). In the following, this approach will be referred to as the *unpredictable* one and will also serve as a comparison basis.

4.5.2.1 Moderate feature models

Here, we compare the SB approach with both the unpredictable approach and SPLCAT. This study is only based on the 2-wise coverage and considers only the moderate FMs.

Experiment setup. To enable a fair comparison, the SB and unpredictable approaches generate sets of configurations of the same size as those provided by SPLCAT. The SB approach is allowed to run for one minute and is performed 10 times per FM. For the 100 generated FMs, the results presented are averaged on all the FM.

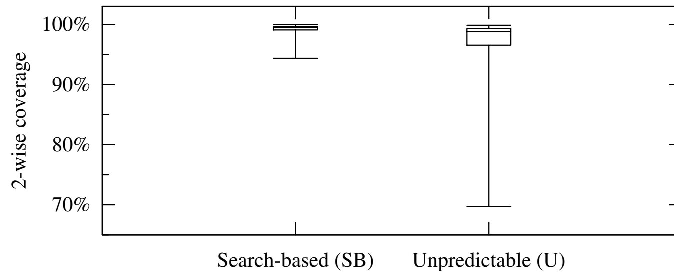
Experiment results. The results are presented in Figure 4.1. SPLCAT is not represented as it always achieves 100% of coverage. The results for the generated FMs are depicted by Figure 4.1a and the results for the real FMs are represented on Figure 4.1b. It appears that the proposed SB approach, as an approximation technique, is close to SPLCAT. Indeed, in the best case, it is able to achieve 100% of 2-wise coverage with only 1 minute of processing time allowed. In the worst case, 95% is achieved on both the generated and real FMs. In addition, the SB approach is much more stable than the unpredictable one, which can drop down to 69% of coverage in the worst case.

Although 100% of coverage might be desirable, the focus of our approach, as explicitly stated in the introduction section, is the partial but scalable t-wise coverage.

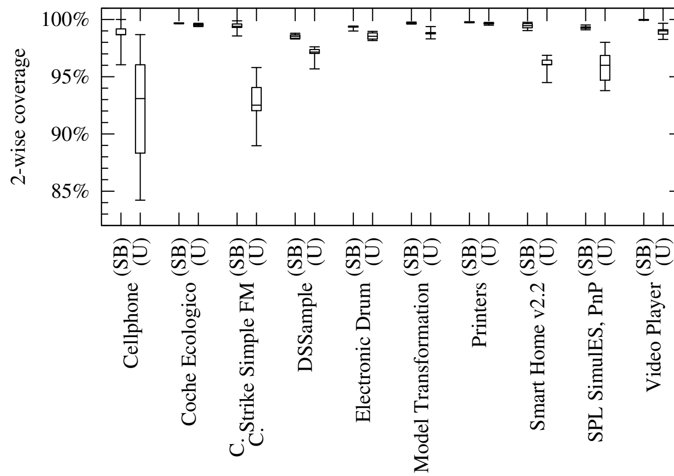
Besides, the performance of SPLCAT varies. For FMs up to 200 features, SPLCAT requires less than a minute. However, it takes around 6.2 minutes for the FMs of more than 200 features, and around 159 minutes for the 1,000 features ones.

Finally, to evaluate whether the difference between the SB approach and the unpredictable one is statistically significant, we followed the guidelines suggested by Arcuri and Briand [AB11]. To this end, we performed a Mann-Whitney U Test[iv]. For each run per FM, we computed the p-value between

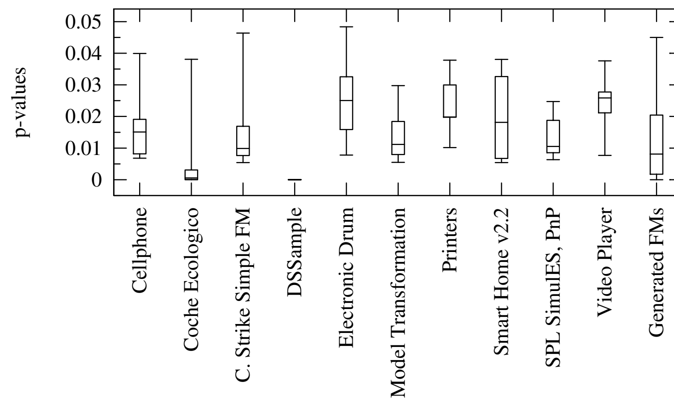
[iv]The Mann Whitney U Test is a non-parametric statistical hypothesis test for assessing whether one of two samples of



(a) Generated feature models (averaged)



(b) Real feature models



(c) Result of the Mann-Whitney U Test between the search-based and the unpredictable approach (equal hypothesis)

Figure 4.1: Configuration generation on the 110 moderate feature models for $t = 2$ (1 minute execution for the search-based approach, 10 runs for each approach. The execution time of the unpredictable approach is few seconds.).

the two approaches. It results in 10 p-values for each FM. Figure 4.1c depicts the distribution of the p-values for the real and generated FMs. For the generated FMs, the p-values are represented all together. Since all the p-values are below the significance level 5%, the difference between the two approaches is considered as statistically different.

4.5.2.2 Large feature models

Scaling to large FMs is a quite difficult task, even for 2-wise. Neither SPLCAT nor the tools we studied (see Section 4.5.1) are able to "scale well to these sizes" [JHF11]. On the contrary, the SB approach efficiently deals with the t-wise combinations where no other approach is able to do so, by producing a partial coverage. Here, we evaluate the t-wise coverage ability of the SB and unpredictable approaches on the large FMs for $t = 2, ..., 6$.

Experiment Setup. To estimate the t-wise coverage of the configurations, we use a process similar as the one used to calculate the t-sets of a FM and described in the beginning of Section 4.5. The sampling process is repeated 10 times per each examined t value ($t = 2$ to $t = 6$) with samples of size 100,000. The SB and unpredictable approaches are executed on all the large FMs to produce 5 times 50 and 100 configurations, with the time restriction of 30 minutes. Another experiment involves the generation of 1,000 configurations and the recording of the coverage over the runs of the SB approach.

Experiment results. The results are recorded in Table 4.4. This table presents the mean coverage achieved with respect to t-wise per FM and per approach. Additionally, it records the standard deviation of these values. A score above 95% with respect to 2-wise is achieved by both the approaches and for all the studied FMs when producing 50 configurations. With respect to 6-wise, scores of 35% to 50% are achieved. By producing 100 configurations, higher scores are achieved for both the approaches. It should be mentioned, based on the standard deviation values recorded in Table 4.4, that a small variation on the achieved coverage is observed. It is a fact indicating that the approaches are quite stable.

Generally, the SB strategy provides a higher coverage compared to the unpredictable approach, especially for high values of t. This is true for all the t-wise coverage measures. Allowing more time to the SB technique should increase the gap with the unpredictable approach since the iterations improve the configuration suites. However, the results are based on the selection of 50 and 100 configurations. Therefore, the maximum difference between the two approaches lies between the coverage of the unpredictable selection and the maximum possible coverage achievable with 50 or 100 configurations. Achieving 100% of t-wise coverage with 50 or 100 configurations seems to be impossible for the large FMs. It is expected that more configurations are needed to achieve 100% of coverage.

To evaluate whether the difference between the two approaches is statistically significant, we perform a Mann-Whitney U Test a the same lines as explained in Section 4.5.2.1. To this end, we applied the following procedure. For each t-value, each number of configurations (50 and 100) each of the 30 runs and each of the 10 t-sets sample, we evaluated the p-value resulting from the test between the search based approach and the unpredictable one. It results in $5 \times 2 \times 30 \times 10$ p-values per FM. Figure 4.2 presents the distribution of these 3000 p-values per FM. The resulting p-values are below the level of significance of 5%, fact indicating that the two approaches are significantly different.

independent observations tends to have larger values than the other. We obtain from this test a probability called p-value which represents the probability that the two samples are equal. It is conventional in statistics to consider that the difference is not significant if the p-value is higher than the 5% level.

Table 4.4: T-wise coverage (%) for the large feature models with 50 and 100 configurations. The search-based approach was allowed to run for 30 minutes. The execution time required by the unpredictable approach is few seconds.

		SB		Unpredictable		SB		Unpredictable	
		50 configurations				100 configurations			
FM	t-wise	Mean	Std.Dev.	Mean	Std.Dev.	Mean	Std.Dev.	Mean	Std.Dev.
eCos	2	99.04	0.10	98.12	0.34	99.60	0.07	99.43	0.17
	3	94.38	0.26	92.28	0.52	97.55	0.14	96.89	0.36
	4	83.40	0.44	80.92	0.53	91.44	0.23	90.41	0.49
	5	67.26	0.49	65.59	0.50	80.02	0.29	79.29	0.51
	6	49.98	0.45	49.43	0.45	64.96	0.29	64.90	0.48
FreeBSD	2	98.89	0.11	97.95	0.27	99.31	0.10	99.17	0.16
	3	95.67	0.16	92.77	0.44	97.89	0.12	96.70	0.30
	4	86.56	0.24	81.88	0.53	93.63	0.20	90.65	0.40
	5	69.97	0.26	65.23	0.48	83.68	0.28	79.29	0.40
	6	50.12	0.22	46.69	0.37	67.57	0.29	63.22	0.36
5000f. gen	2	95.92	0.12	94.93	0.29	98.30	0.08	97.83	0.19
	3	85.94	0.20	84.04	0.33	92.31	0.16	91.19	0.25
	4	70.50	0.21	68.24	0.30	80.86	0.25	79.22	0.25
	5	52.89	0.18	50.87	0.25	65.39	0.25	63.65	0.25
	6	36.63	0.18	35.18	0.23	48.92	0.25	47.44	0.23
Linux	2	97.74	0.16	97.03	0.23	98.74	0.09	98.46	0.15
	3	93.03	0.21	91.86	0.28	96.03	0.13	95.47	0.21
	4	82.67	0.24	81.25	0.27	90.28	0.18	89.48	0.22
	5	65.77	0.23	64.50	0.25	79.33	0.20	78.40	0.21
	6	46.48	0.18	45.63	0.20	63.08	0.21	62.25	0.23

Table 4.5: 6-wise coverage and fitness evolution over time for the search-based approach on the large feature models with 1,000 configurations.

	0 run (=unpred.)		5,000 runs		10,000 runs		15,000 runs	
	Coverage	Fitness	Coverage	Fitness	Coverage	Fitness	Coverage	Fitness
eCos	94.191%	271,880	94.225%	286,304	94.263%	288,039	95.343%	288,818
FreeBSD	76.236%	294,184	76.395%	299,962	76.465%	300,892	76.494%	301,634
Generated FM	82.986%	258,763	84.492%	263,243	84.605%	263,974	84.778%	264,362
Linux	89.411%	296,661	90.404%	298,709	90.640%	299,114	90,671%	299,363

Table 4.5 records the coverage achieved by the SB approach each 5,000 runs repetitions for 1,000 configurations with respect to 6-wise. Here, we observe that a higher level of coverage is achieved with more configurations. For instance, the SB approach achieves 90,671% of 6-wise coverage for the Linux FM. It also shows, as it can be expected for a SB approach, that allowing more processing time to the approach allows reaching a higher coverage. Indeed, at each 5,000 runs, the coverage recorded is higher than the previous one. Here, the unpredictable approach, represented by the "0 run", is also the initialization stage of the SB strategy (Alg. 3, lines 5 to 10). For example, considering the eCos FM, 94.191% of 6-wise coverage is achieved at the initialization. After 15,000 runs, it is 95.343%, which represents ≈ 2.475744E15 additional 6-sets covered compared to the unpredictable approach. The number of valid t-sets is extremely high (see Table 4.2) and thus, a small increase in the coverage represents a high increase in the number of additional valid t-sets covered. Finally, the 15,000 runs require about 10 to 20 hours of processing time per FM.

4.5.2.3 Fitness function

So far, the presented results suggest that the SB approach is effective and able to scale to large FMs. Scalability is reached thanks to the ability of the similarity fitness function to mimic the t-wise coverage. To illustrate this fact, Table 4.5 records the fitness function values with respect to 6-wise coverage for the large FMs as the SB approach evolves. It shows that the fitness increases with the coverage over the runs of the approach. The same trend holds for all the FMs and values of t considered in this study. Figure 4.3 illustrates the correlation between the fitness and the t-wise coverage for the Linux FM. Therefore, the assessment of a configuration suites can be performed without computing any t-set, thanks to the fitness function. Recall that computing the t-sets requires vast computational resources (Section 4.3.1, Eq. 4.1).

4.5.2.4 Research question 2 summary

The configuration generation experiments emphasize the following outcomes. First, **the similarity heuristic and the fitness function driving the approach form an efficient guide toward the configurations selection.** The SB configuration generation mimics the t-wise coverage, does not depend at all on t and thus, it avoids the combinatorial explosion due to the combinations of t features. Second, **the proposed technique is the first one, to the authors' knowledge, which scales well to large FMs while achieving a decent level of t-wise coverage** (depending on the number of configurations desired). Finally, in addition to be a close approximation of SPLCAT, it is more flexible than the latter as it allows specifying the processing time and the number of desired

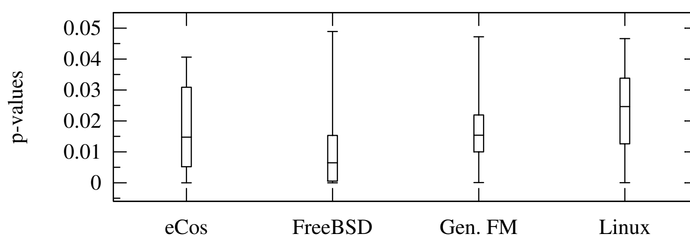


Figure 4.2: Result of the Mann-Whitney U Test between the search-based and the unpredictable approach for $t = 2...6$ on the four large feature models (equal hypothesis).

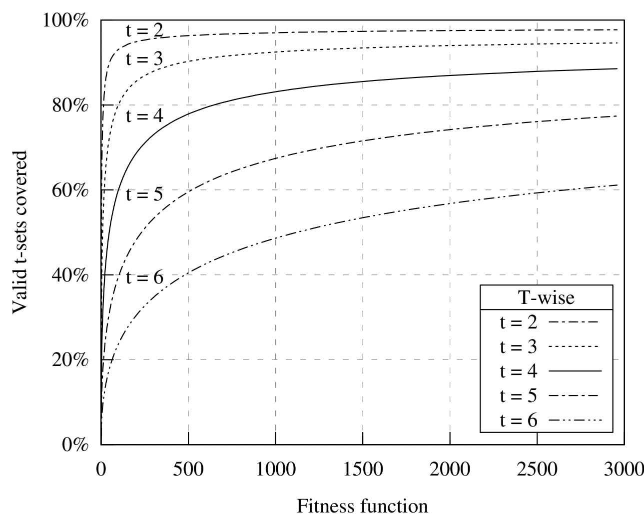


Figure 4.3: Fitness function correlation with t-wise coverage for the Linux kernel feature model.

configurations. These are characteristics conforming to an industrial context where the testing process is subjected to budget constraints.

4.5.3 Configuration prioritization assessment (research questions 3 & 4)

This part evaluates the proposed prioritization approaches. To this end, we compare them with a prioritization technique based on interaction coverage from Bryce and Memon [BM07]. This technique is commonly used in CIT studies and it is based on t-wise interaction coverage. Since similarity forms an alternative to the t-wise evaluation, it is natural to compare with it. In the remainder of this chapter, we refer to this approach as Interaction-based.

The first experiment focuses on $t = 2$ for the moderate FMs, due to the limitations of SPLCAT (see Section 4.5.2 and 4.5.2.1). The second experiment demonstrates the ability of the similarity-based approaches to scale to any t value for the large FMs. This second part does not consider SPLCAT and the Interaction-based approach [BM07] given their inability to scale, as demonstrated by our results (see Section 4.5.3.1). Finally, to compare the prioritization approaches, the area under curve is evaluated.

4.5.3.1 Moderate feature models

This part of the experiments compares our prioritization techniques to SPLCAT and the Interaction-based approach for $t = 2$. SPLCAT does not provide an independent prioritization approach as we do but it tries to cover the maximum of 2-sets each time a configuration is added, effectively implementing the greedy heuristic for prioritization [YH12]. The resulting configurations can thus be considered as ordered for covering faster the highest amount of 2-sets.

Experiment setup. For each moderate FMs, three different sets of configurations are used to apply the prioritization techniques. The first set is the configuration suites produced by SPLCAT (Case I). The second one is a configuration suites of $m = \frac{\#features}{2}$ configurations, selected with the unpredictable method (Case II). Finally, the last set is composed of the configurations generated by SPLCAT plus the same amount of configurations selected by the unpredictable method (Case III). Using these different sets allows ensuring that the prioritization approaches are relevant whatever the nature of the configurations.

All these sets of configurations are randomized before executing the prioritization techniques. This practice ensures that our approaches are independent of the original order. On each of the three cases and for each FM, a random prioritization is averaged 10 times. Cases II and III are independently repeated 10 times to avoid any bias from the initial configuration suites. For each approach and each independent repetition, the execution time is recorded.

Experiment results. Table 4.6 presents the area under curve for each case and technique. Recall that a higher surface value indicates a better prioritization. With respect to Table 4.6 and focusing on Case I, we observe the following ordering: Random < Local Maximum Distance < SPLCAT < Global Maximum Distance ≈ Interaction-based. For Case II and Case III, the order Random < Local Maximum Distance < Global Maximum Distance < Interaction-based is observed. However, the Global Maximum Distance approach performs almost equally as the Interaction-based one (difference of 0.01 in the area under curve). It shows the ability of the similarity heuristic to mimic the t-wise coverage. In addition, it also performs better than the Random and Local Maximum Distance.

Table 4.6: Prioritization results: area under curve (scale 1:1,000).

Technique \ t	Case I	Case II	Case III	Case IV / 100 confs.					Case IV / 500 confs.					Case V / 100 confs.					Case V / 500 confs.				
	2	2	2	2	3	4	5	6	2	3	4	5	6	2	3	4	5	6	2	3	4	5	6
Random	7.99	7.74	8.93	9.23	8.34	7.10	5.57	4.05	49.06	47.69	45.09	40.88	35.22	8.65	7.33	5.67	4.02	2.67	47.47	44.70	40.42	34.40	27.40
Local Max. Dist.	8.33	7.89	9.07	9.28	8.43	7.17	5.61	4.07	49.16	47.77	45.15	40.97	35.32	8.89	7.68	6.13	4.45	3.03	47.76	45.28	41.44	36.14	29.55
Global Max. Dist.	8.43	8.02	9.19	9.33	8.48	7.22	5.65	4.11	49.23	47.92	45.46	41.32	35.66	9.06	8.00	6.56	4.91	3.37	48.15	46.19	42.95	38.03	31.71
Interaction-based	8.43	8.03	9.20																				
SPLCAT	8.37																						

Figure 4.4 illustrates this behavior. For each case and category of FM (real and generated), the results are averaged on all the FMs of the category by normalizing the number of configurations selected from 0 to 100%. For instance, with respect to Case I and the generated FMs (Figure 4.4a), the Global Maximum Distance prioritization approach enables covering more than 90% of the 2-set with only 30% of the configurations. On the contrary, the random prioritization needs around 50% of the configurations. For Case II and Case III, the same trends are observed. These results emphasize that the prioritization techniques are either able to perform similarly (Local Maximum Distance) or

Table 4.7: Prioritization results: execution time in milliseconds.

	Case I	Case II	Case III	Case IV/100 confs.	Case IV/500 confs.	Case V/100 confs.	Case V/500 confs.
Local Max. Dist.	241.3	263.4	288.1	1,693	45,437	1,602	46,299
Global Max. Dist.	250.1	271.2	299.4	1,764	47,310	1,698	48,267
Interaction-based	45,256	61,371	111,864				

better (Global Maximum Distance) as both the SPLCAT and the interaction-based approach.

Regarding the execution time, consider Figure 4.5. It shows the average execution time for the considered prioritization approaches on the real FMs (Figure 4.5a) and on the generated FMs (Figure 4.5b). SPLCAT is not considered as it is a configuration generation approach. From these figures, it is clear that the Interaction-based technique has difficulties to scale. This is due to the expensive computation of the t-wise interactions (see Section 4.3.1, Equation 4.1). For instance, consider the generated FMs (Figure 4.5a). For models of 200 features, it requires around 50,000 milliseconds. For FMs of 500 features, the execution time increases to more than 10^6 milliseconds.

On the contrary, the Local and Global Maximum Distance approaches are significantly less impacted by the complexity of the FM. Finally, Table 4.7 shows the execution time for the different cases. Overall, the Global Maximum Distance approach provides a little overhead compare to the Local Maximum Distance. Given the fact that the Global Maximum Distance performs similarly to the Interaction-based technique with a considerably lower computational overhead, its use is advisable.

4.5.3.2 Large feature models

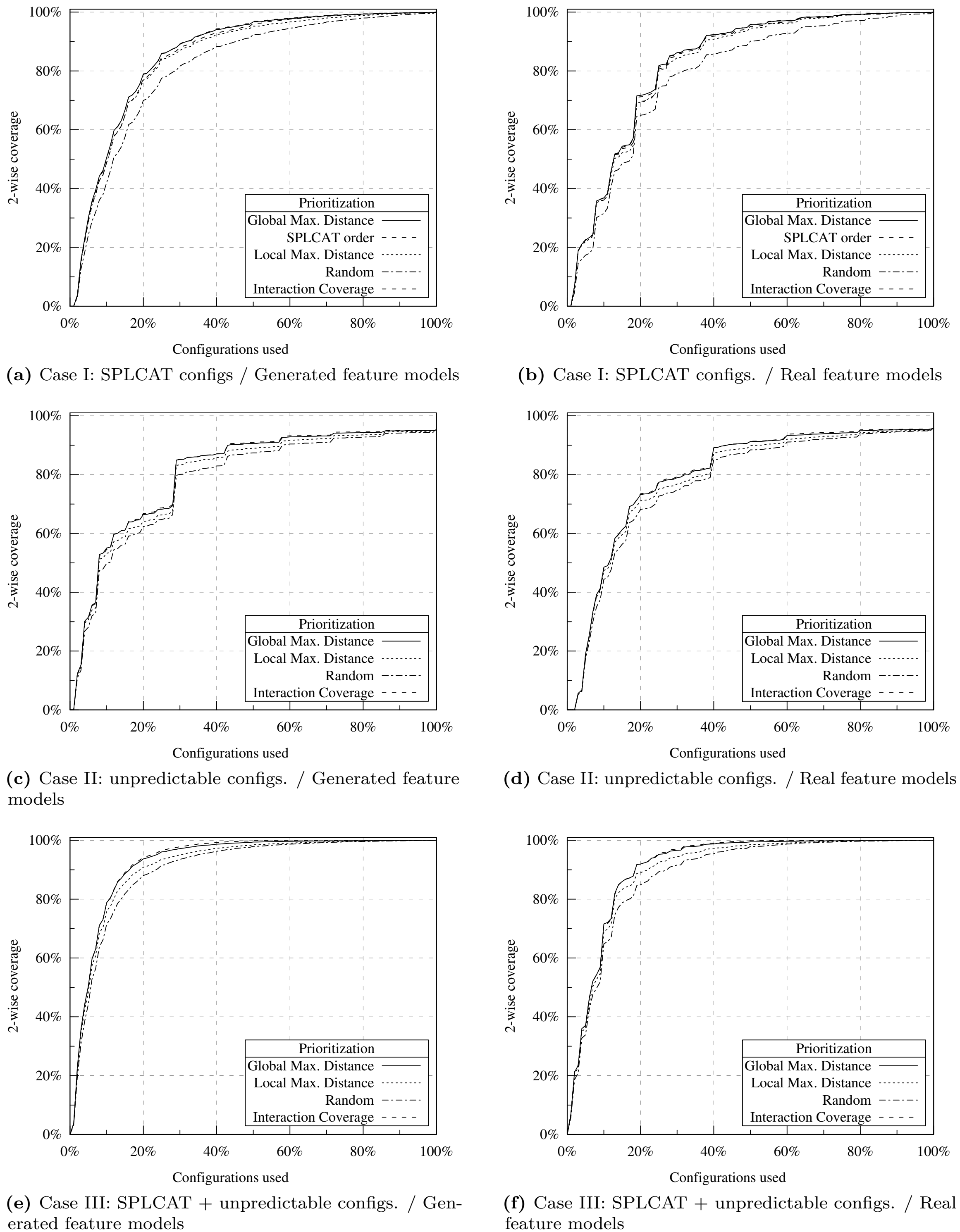
This part of the study assesses the Global Maximum and Local Maximum Distance prioritizations on the large FMs for $t = 2, ..., 6$. The Interaction-based approach is not considered given its difficulty to handle moderate size FMs, as shown in the previous section.

Experiment Setup. We generate two sets of 100 and 500 configurations containing dissimilar configurations (Case IV) and two sets of the same sizes containing half similar and dissimilar configurations (Case V). We choose these two kinds of sets of configurations since the prioritization approaches are similarity-driven and can thus be influenced by the nature of the used sets. Indeed, applying these approaches on sets containing dissimilar configurations can be less effective than applying them on sets containing similar configurations. We randomize each configuration suites and execute the Local Maximum Distance and Global Maximum Distance prioritizations on each of them. We also produce 10 random orderings to compare with our approaches. This practice shows that the prioritization techniques are not affected by random orders.

Experiment results. As for the results presented in Section 4.5.3.1, we evaluate the area under curve. The results are recorded in Table 4.6, Cases IV and V. Random is averaged on 10 runs for each value of t. The presented values are averaged on the 4 large FMs. We observe the following ordering for both Case IV and Case V: Random < Local Maximum Distance < Global Maximum Distance. Thus, the prioritizations approaches are relevant for finding the dissimilarities in the sets containing both similar and dissimilar configurations. The Global Maximum Distance prioritization tends to be the most relevant approach.

As expected, when configurations are already dissimilar (Case IV), the gain is lesser than when the configuration suites is any (Case V). Additionally, Figure 4.6 presents the t-wise coverage difference between the Global Maximum Distance prioritization and the random ordering for 500 configurations, averaged on the 4 FMs. For Case IV (Figure 4.6a) and $t = 4$, 3% of difference is observed with 30

configurations selected. For Case V (Figure 4.6b), 14% of difference is observed with 100 configurations for $t = 6$.



(a) Case I: SPLCAT configs / Generated feature models

(b) Case I: SPLCAT configs. / Real feature models

(c) Case II: unpredictable configs. / Generated feature models

(d) Case II: unpredictable configs. / Real feature models

(e) Case III: SPLCAT + unpredictable configs. / Generated feature models

(f) Case III: SPLCAT + unpredictable configs. / Real feature models

Figure 4.4: Prioritization on the moderate size feature models for $t = 2$.

4.5.3.3 Research questions 3 & 4 summary

The experiments conducted for the prioritization bring out the following conclusions. First, the Global Maximum Distance performs similarly to the Interaction-based approach. However, the **Global Maximum Distance approach is significantly faster and less sensitive to the complexity of the FM than the Interaction-based one**. Thus, it forms a scalable approach for prioritizing configurations. Second, **the most relevant configurations contributing to t-wise coverage are the most dissimilar ones**. This is enabled by the similarity heuristic. Finally, **the proposed prioritization approaches are able to prioritize any configuration suites**, by looking for the dissimilarities. This is performed without computing any t-sets and regardless of the value of t.

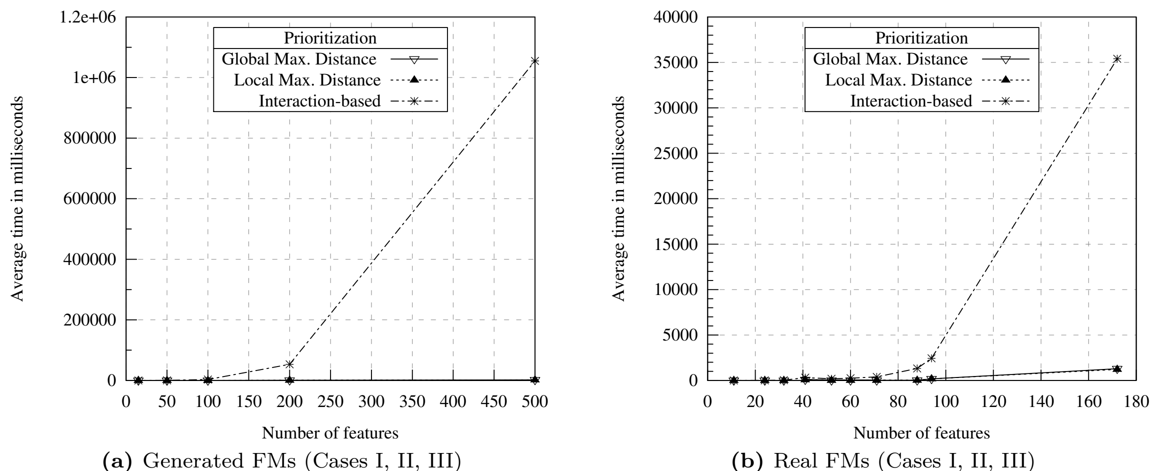
4.6 Discussion

This section first discusses the interaction fault detection ability of configuration suites. Then, it presents practical implications and further applications of our approaches. Finally, the limitations of our techniques and the threats to the validity of the conducted experiments are highlighted.

4.6.1 Detecting t-wise interaction faults

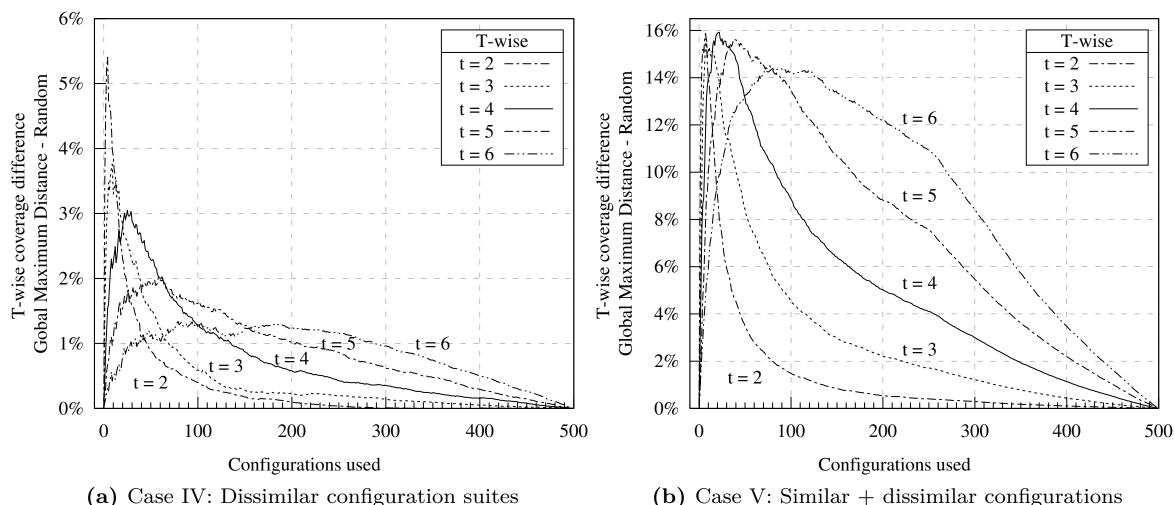
Failure due to interactions are difficult to detect as they occur when several features are involved together. Generally, each feature can be tested independently, e.g., using unit testing. Highlighting t-wise faults is more difficult. Arcuri and Briand [AB12] established the lower bound for the probability of a random test suite to trigger at least one failure related to t-wise. However, this bound is only valid in the context of CIT without constraints.

In our context, features are constrained and the above-mentioned results cannot be applied directly. Providing theoretical results, such as those of Arcuri and Briand [AB12] is not possible in the presence of constraints. This is due to the fact that constraints are specific to each FM. Therefore, in our



(a) Generated FMs (Cases I, II, III) **(b)** Real FMs (Cases I, II, III)

Figure 4.5: Execution time for the prioritization on the moderate size feature models for $t = 2$. Each approach has been performed 10 times per feature model.



(a) Case IV: Dissimilar configuration suites

(b) Case V: Similar + dissimilar configurations

Figure 4.6: Global Maximum Distance VS Random prioritization on the four large feature models.

work, we turn to empirical analysis. In order to complete the experiments with reasonable resources, we restrict the experiments to t-wise interaction faults for $t = 2$ to 6. If we consider that all the t-wise interactions of the SPL have the same probability to trigger a fault and that concrete test cases derived from a selected configuration expose all the feature interaction faults that are present in this configuration, then the probability that a fault is found by a configuration suite can be represented by the t-wise coverage [AB12]. This probability is calculated by summing the t-wise coverage for all the examined t values ($t = 2, ..., 6$).

Figure 4.7 depicts the probability of finding faults for the large FMs for all the t-values in the context of prioritization. This probability is obtained by summing the probabilities for all the FMs. Compared to a random ordering, a difference of about 15% in the probability to find a fault can be observed with around 100 configurations. It means that with the first 100 configurations proposed by our approach, we reach a probability of finding a fault equal to 57% whereas a random ordering would need more than 200 configurations to reach the same probability.

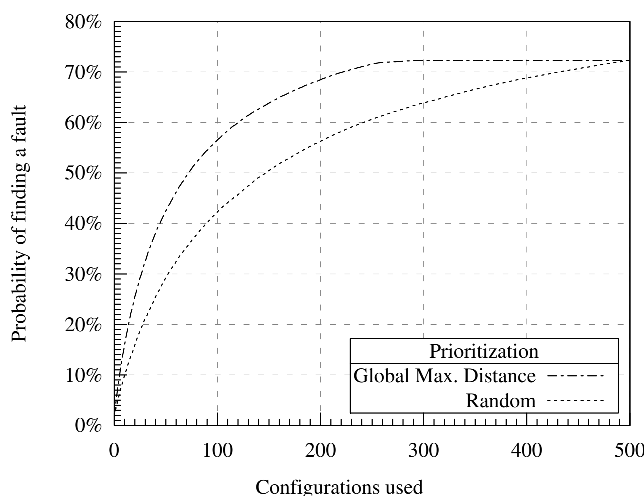


Figure 4.7: Estimation of the interaction fault detection rate of the Global Maximum Distance and Random prioritizations for all the t-values. It uses the configurations resulting from the prioritization experiment on the four large feature models (Case V).

Regarding the configuration generation, Table 4.8 presents the estimation of the fault detection rate achieved by the SB and the unpredictable approaches on the 4 large FMs, for $t = 2, ..., 6$. For 50

Table 4.8: Estimation of the interaction fault detection rate on the 4 large feature models for $t = 2, ..., 6$ for the search-based and unpredictable approaches.

	50 confs.		100 confs.	
	SB	Unpredictable	SB	Unpredictable
eCos	78.81%	77.28%	86.70%	86.19%
FreeBSD	80.24%	76.90%	88.41%	85.80%
5000f. gen.	71.26%	68.31%	71.75%	71.26%
Linux	77.14%	76.05%	85.49%	84.81%
avg	76.8625%	74.635%	83.0875%	82.015%

configurations, the SB approach yields an estimated interaction fault detection rate of 76.86% and the unpredictable one a rate of 74.63%. Thus, an average difference of more than 2% is observed between the two techniques with only 50 configurations.

4.6.2 Practical implications

While using existing tools such as SPLCAT, we realized that existing approaches which implement a covering array technique already perform a prioritization with respect to t-wise interactions. This is due to the construction of these techniques which try to cover the maximum amount of t-sets with each new configuration. As a result, the outcome of our prioritization techniques on small or moderate size FMs is limited, as existing approaches already perform a prioritization. However, even in this case, our techniques perform better than the existing ones. The benefit of our techniques is more evident in the context of large SPLs, where other tools cannot work. In addition, our prioritization methods can operate on any configuration suites. They can therefore be used in combination with existing approaches and tools.

Our generation approach requires a FM. It can work on any FM, regardless of its complexity. As a result, the system abstracted by the FM has no impact on our approach. When the available model is not a FM, and if the model can be translated to a Boolean one, then our approach can also be applied. Indeed, it is still possible in that case to employ transformation rules, such as [FPDN05] in order to transform the model to a Boolean one. If such a transformation is not possible, our technique can be combined with a SMT solver to handle non-Boolean models. Finally, our technique does not rely on code or existing test cases. Such artifacts may not be available at an early stage of the system development or hard to analyze directly due to their size and complexity. As a result, such cases are well suited for applying our approach.

The number of configurations to generate and the amount of time allowed to generate them are parameters of our SB approach. These parameters aims at making the testing process more flexible. Indeed, existing approaches [JHF11] generate all the configurations necessary to cover all the t-sets. The problem is that they may take a large amount of time to perform this full coverage and they may generate too many configurations. To the authors' knowledge, our approach bestows a unique feature which aims at maximizing the t-wise coverage for the specified amount of configurations, given the specified amount of time. As a result, it gives a partial coverage but also makes the testing process more practical for large SPLs. In any cases, for large FMs, it is not possible to evaluate all the t-wise interactions.

Besides, our approach scales to large FMs. While using constraint solvers, we observed that solving constraints is a time consuming task and thus an obstacle to scalability. In addition, calculating the t-wise coverage is difficult for large FMs since all the t-sets of the configurations have to be considered. Our generation approach uses a SAT solver only for generating configurations satisfying the constraints of the FM. The prioritization and generation techniques are driven by a similarity heuristic which mimics t-wise coverage and does not require to compute any combination of feature. As a consequence, one strength of the proposed approach is that it maximizes the coverage for any t

value. On the contrary, existing approaches focus only on a given t-value. This results in maximizing the fault detection up to t while leaving aside the higher strengths of t [AB12]. Thus, our approach has a clear advantage over existing ones.

4.6.3 Further applications

The propositions made in this chapter have potential application to other possible issues related to CIT or SPL testing. For example, considering the evolution of a SPL over time [RE12], our approach can also be relevant. In that context, different versions of the FM exist (the original and the evolved FMs). Each version represent a model of the SPL. Therefore, taking into account the evolution over time implies modifying the way the distances are computed. The aim is to focus on the features that have changed or that have been added to the evolved version of the SPL. Thus, by defining a distance measure which ignores the unchanged features from the calculation, the proposed approach is generate and prioritize configurations over the interactions of the modified or new features.

Along the same lines, other approaches [SPF12, KBK11, JHF$^+$12b] attribute different importance to t-wise combinations. This practice can reduce the complexity of the problem since only the most important combinations are considered. However, our approach also targets on early development stage where no code or system is available. Furthermore, even in the case where the system code is available, these approaches face the usual pitfalls of dynamic analysis. Thus, scalability issues, problems of handling system calls or memory constructs restrict the application of such approaches on large scale systems. In addition, our approach is somehow orthogonal to these techniques due to the generation and prioritization. In order to take into account the relative importance of feature combinations, there is a need to assign weights representing the importance of the interactions. These weights can be assigned based on the use of dynamic approaches. Our approach can handle weight by defining an appropriate distance measure. In this work, we consider all the features and features combination as equal. The importance of features or combination of features is a worth studying problem which has been left open for further research, adding one objective to be considered by the prioritization algorithm [HPP$^+$13c].

4.6.4 Limitations

The first limitation of our configuration generation approach is that it does not provide a full coverage. Indeed, we perform a partial but scalable t-wise coverage. In other words, we try to maximize the t-wise coverage achieved for a given number of configurations, within the specified amount of time provided. This is not a problem as evaluating all the t-wise interaction is difficult for very large SPLs. In addition, in an industrial context, the testing budget is typically limited, preventing all the SPs of the SPL from being tested. Thus, a smaller to 100% coverage is going to be achieved. The second limitation is that our approach works only with FMs. It may also work with other models that can be translated to Boolean ones. Alternatively, a SMT solver can be used to satisfy non-Boolean constraints. Finally, we depend on the scalability of constraint solvers and the overhead they induce. We believe that this is not important as we did not find any concrete FM raising such an issue. We acknowledge this as a potential limitation for the cases of FMs significantly larger than the studied ones.

4.6.5 Threats to validity

Although we used various FMs, there is an *external validity* threat. Indeed, we cannot ensure that the proposed strategies will provide similar results on different sets of FMs (larger or more constrained). To reduce this threat, we used a relatively large set of 114 FMs of different sizes, combined real and generated FMs to cope with a variety of situations. Additionally, potential errors in our implementation could affect the presented results and lead to *internal validity* threats. To diminish these threats, we divided the implementation into sub stages to have a better control on each of the steps composing the proposed approaches. The comparison with existing tools also gave us confidence in our implementation. In addition, in order to enable reproducibility and to reduce the above-mentioned threats, we made our implementation and the experiment data publicly available. Besides that, to prevent as possible a *construct validity* threat, we sampled each technique on 10 runs.

Another threat concerns the identification of faulty interactions by the actual testing of a concrete SP. It is assumed that testing a SP ensures revealing the interaction faults that it contains. While this holds, it is a common assumption made by all the CIT approaches. CIT techniques require to cover at least once each t-wise interaction, supposing that executing the configuration suite will effectively reveal the faulty interactions. Additionally, this assumption is in line with the structural testing (code coverage) approaches. Branch coverage forms a testing requirement in many software standards, e.g., [oST]. However, covering branches or statements assumes that executing parts of the code will trigger the faults that they contain [FHLS98]. Besides, it should be clear that to reveal a faulty interaction, a test should exercise this interaction. Suppose that a configuration suite covers $x\%$ more t-wise interactions than another one. Then, it is guaranteed that the other configuration suite will miss all the faults contained in these x interactions. Finally, recent studies [KWG04, PYCH13] demonstrate the correlation between t-wise and fault detection.

4.7 Conclusions

T-wise testing aims at finding faulty feature interactions. However, full t-wise testing is hard and scalability is an issue: no approach is able to deal with high values of t (≥ 3) for large SPL FMs in a reasonable amount of time (in days). Moreover, there is no suitable technique supporting the generation of a fixed number of configurations, according to a limited budget. This chapter tackled these problems by proposing (a) approaches to prioritize configurations while maximizing the t-wise coverage and (b) a scalable and flexible SB technique to generate configurations under budget and time constraints for large FMs.

Our experiments, performed on 100 artificially generated and 14 real FMs from $t = 2$ to $t = 6$ show the feasibility and the scalability of our solutions. We managed to deal with the largest FMs available, such as the Linux kernel ($\approx$ 7,000 features, $\approx$ 200,000 constraints and $\approx$ 8.71E21 valid 6-sets) with up to 90.671% of 6-wise coverage achieved with 1,000 configurations. Thus, by enabling a partial but scalable t-wise coverage and by introducing flexibility in the testing process, our approaches pave the way to a potentially t-unrestricted CIT. Finally, our implementations and the experimental data are publicly available at `http://research.henard.net/SPL/`.

5

A mutation-based approach for generating software product line configurations

In the previous chapter, we presented a scalable technique for generating software product line configurations for combinatorial testing. This chapter introduces a mutation-based approach for generating software product line configurations, which forms an alternative to the traditional combinatorial interaction testing criterion.

This chapter is based on the work that has been published in the following paper:

- Christopher Henard, Mike Papadakis, and Yves Le Traon. Mutation-based generation of software product line test configurations. In Claire Le Goues and Shin Yoo, editors, *Search-Based Software Engineering*, volume 8636 of *Lecture Notes in Computer Science*, pages 92–106. Springer International Publishing, 2014

Contents

5.1 Introduction **58**
5.2 The mutation-based configuration generation approach **58**
5.2.1 Creation of mutants of the feature model 59
5.2.2 The search-based process 59
5.3 Experiments **61**
5.3.1 Approach assessment (research question 1) 62
5.3.2 Comparison with random (research question 2) 64
5.4 Threats to validity **66**
5.5 Conclusions **67**

5.1 Introduction

CIT is widely used to reduce the difficulty of testing SPLs. While this criterion is effective for disclosing bugs [KWG04, PYCH13], recent work has shown mutation as a promising alternative to the CIT criterion, also correlating with fault detection [PHT14] for existing test suites.

The use of mutation. In this chapter, mutation is used to produce defective versions of the FM. As introduced in Chapter 2, a mutant is an altered version of the rules defining the legal feature associations. Such mutants are useful as they represent faulty implementations of the FMs that should be tested. Thus, while CIT measures the number of feature interactions of the FM exercised by the test suite, mutation measures the number of mutants detected by the test suite. However, and despite the potential benefit of mutation, there is no approach with the purpose of generating configurations for SPL with respect to the mutation criterion.

Contributions of this chapter. This chapter devises the first approach which generates SPL configurations using mutation of the FM. Since the SPL configuration space is too large to be exhaustively explored, we introduce a SB technique based on the (1+1) Evolutionary Algorithm (EA) [DJW02, LY14] in conjunction with a constraint solver in order to only deal with valid configurations. In order to guide the search towards the detection of mutants, four search operators are proposed to both add and remove configurations from the test suite. The proposed approach solves the challenge of generating a test suite with respect to the mutation criterion. Experiments on 10 FMs show the ability of the proposed approach to generate test suites while with the purpose of mutation.

In brief, the present chapter provides the following insights:

- We propose the first mutation-based approach for selecting configurations for SPLs. It is based on four search operators.
- We conduct an experiment on 10 FMs.

The remainder of this chapter is organized as follows. Section 5.2 describes the approach itself. Section 5.3 presents the conducted experiments and Section 5.4 discusses threats to their validity. Finally, Section 5.5 concludes the chapter.

5.2 The mutation-based configuration generation approach

The approach for generating configurations starts by creating mutants of the SPL FM. Then, a SB process based on the (1+1) Evolutionary Algorithm (EA) [DJW02, LY14] makes use of both the FM and the mutants to produce a set of configurations. The (1+1) EA is a hill climbing approach which has been proven to be effective in several studies [HPP+14, HM10] and been used in the previous chapter. The overview of the approach is depicted in Figure 5.1. The following sections describe the different steps of the approach.

5.2.1 Creation of mutants of the feature model

The first step of the approach creates altered versions of the FM. Each altered version is called a mutant and contains a defect within the boolean formula of the FM. For instance, the two following mutants are produced from the FM example of Figure 2.1:

$$\begin{aligned} M_1 = {} & \overset{\blacktriangledown}{\overline{f_1}} \wedge (\overline{f_2} \vee f_1) \wedge (\overline{f_1} \vee f_2) \wedge (\overline{f_3} \vee f_1) \wedge (\overline{f_1} \vee f_3) \wedge (\overline{f_4} \vee f_1) \wedge (\overline{f_5} \vee f_1) \wedge (\overline{f_1} \vee f_5) \wedge (\overline{f_6} \vee f_3) \wedge (\overline{f_7} \vee f_3) \wedge \\ & (\overline{f_3} \vee f_6 \vee f_7) \wedge (\overline{f_8} \vee f_5) \wedge (\overline{f_9} \vee f_5) \wedge (\overline{f_5} \vee f_8 \vee f_9) \wedge (\overline{f_8} \vee \overline{f_9}) \wedge (\overline{f_7} \vee f_4) \wedge (\overline{f_4} \vee \overline{f_8}) \wedge (\overline{f_9} \vee f_4). \end{aligned}$$

$$\begin{aligned} M_2 = {} & f_1 \wedge (\overline{f_2} \overset{\blacktriangledown}{\wedge} f_1) \wedge (\overline{f_1} \vee f_2) \wedge (\overline{f_3} \vee f_1) \wedge (\overline{f_1} \vee f_3) \wedge (\overline{f_4} \vee f_1) \wedge (\overline{f_5} \vee f_1) \wedge (\overline{f_1} \vee f_5) \wedge (\overline{f_6} \vee f_3) \wedge (\overline{f_7} \vee f_3) \wedge \\ & (\overline{f_3} \vee f_6 \vee f_7) \wedge (\overline{f_8} \vee f_5) \wedge (\overline{f_9} \vee f_5) \wedge (\overline{f_5} \vee f_8 \vee f_9) \wedge (\overline{f_8} \vee \overline{f_9}) \wedge (\overline{f_7} \vee f_4) \wedge (\overline{f_4} \vee \overline{f_8}) \wedge (\overline{f_9} \vee f_4). \end{aligned}$$

In M_1, a literal has been negated whereas in M_2, an operator OR has been replaced by an AND one. It should be noted that the proposed approach is independent from the way the mutants have been created and from the changes they operate compared to the original FM.

5.2.2 The search-based process

Once the mutants are created, the SB process starts to generate a set of configurations. The different steps of the approach are described in Algorithm 4 and detailed in the following. First, an initial population is created and its fitness is evaluated (line 1 and 2). Then, the population is evolved (line 3 to 10): search-operators try to improve the population by adding or removing configurations.

5.2.2.1 Individual

An individual I or potential solution to the problem is a configuration suite of k connfigurations that are satisfying the FM constraints: $I = \{C_1, ..., C_k\}$.

5.2.2.2 Population

The population P is composed of only one individual: $P = \{I\}$.

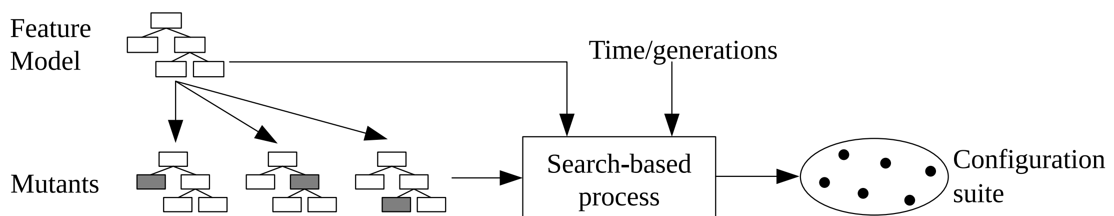


Figure 5.1: Overview of the mutation-based approach for generating configurations.

5.2.2.3 Initial population

The individual of the initial population is initialized by generating randomly a valid configuration by using a SAT solver.

5.2.2.4 Fitness evaluation

The fitness f of an individual I is calculated by evaluating how many mutants are not satisfied by at least one of the configurations of I, i.e., the MS. More formally, if we denote as $M = \{M_1, ..., M_m\}$ the m mutants of the FM, the fitness f of an individual I is evaluated as follows:

$$F(I) = \frac{|\{M_i \in M \mid \exists C_j \in I \mid C_j \text{ does not satisfy } M_i\}|}{m} = \text{MS},$$

where $|A|$ denotes the cardinality of the set A. It should be noted that all the configurations considered are satisfying the FM constraints since they belong to I.

5.2.2.5 Search operators

The approach makes use of four search operators that operate on an individual I. The operators are divided into two categories: operators that *add* a new configuration and operators that *remove* a configuration. The operators are depicted in Figure 5.2.

- **Add a random configuration**. This operator is presented in Figure 5.2a. It adds to the considered individual a configuration randomly chosen from the space of all the configurations of the FM .
- **Remove a random configuration**. This operator is depicted in Figure 5.2b. It randomly removes a configuration from the individual.
- **Smart add of a configuration**. This operator is presented in Figure 5.2c. First, the altered constraints of the mutants are collected. Then, for each constraint, the number of configurations from I that do not satisfy it is evaluated. This can be view as a mutant constraint score. Then, using this score, a proportionate selection is performed in order to choose one of these constraints. The idea is to promote the constraint that is the less not satisfied by the configurations of I. Then, the operators tries to select a configuration which is at the same time satisfying the FM

Algorithm 4 Mutation-based generation of configurations

1: Create an initial population P with one individual I : $P = \{I\}$ containing one configuration
2: Evaluate the fitness f of I : $f = F(I)$
3: **while** budget (time, number of generations) **do**
4: Select a search operator with a probability p
5: Generate a new individual I' using the selected search operator
6: Evaluate the fitness $f' = F(I')$
7: **if** $f' \geq f$ **then**
8: $I = I'$
9: **end if**
10: **end while**
11: **return** I

and the negation of the selected constraint. Doing so will result in a configuration that is able to violate a clause of the mutant and thus do not satisfy it.

- **Smart remove of a configuration**. This operator is illustrated in Figure 5.2d. For each configuration of I, it is evaluated the number of mutants that are not satisfied. This can be view as a configuration score. Then, using this score, a proportionate selection is performed in order to choose which configuration to remove from I. The idea is to promote the removal of configurations that are not satisfying the less amount of mutants.

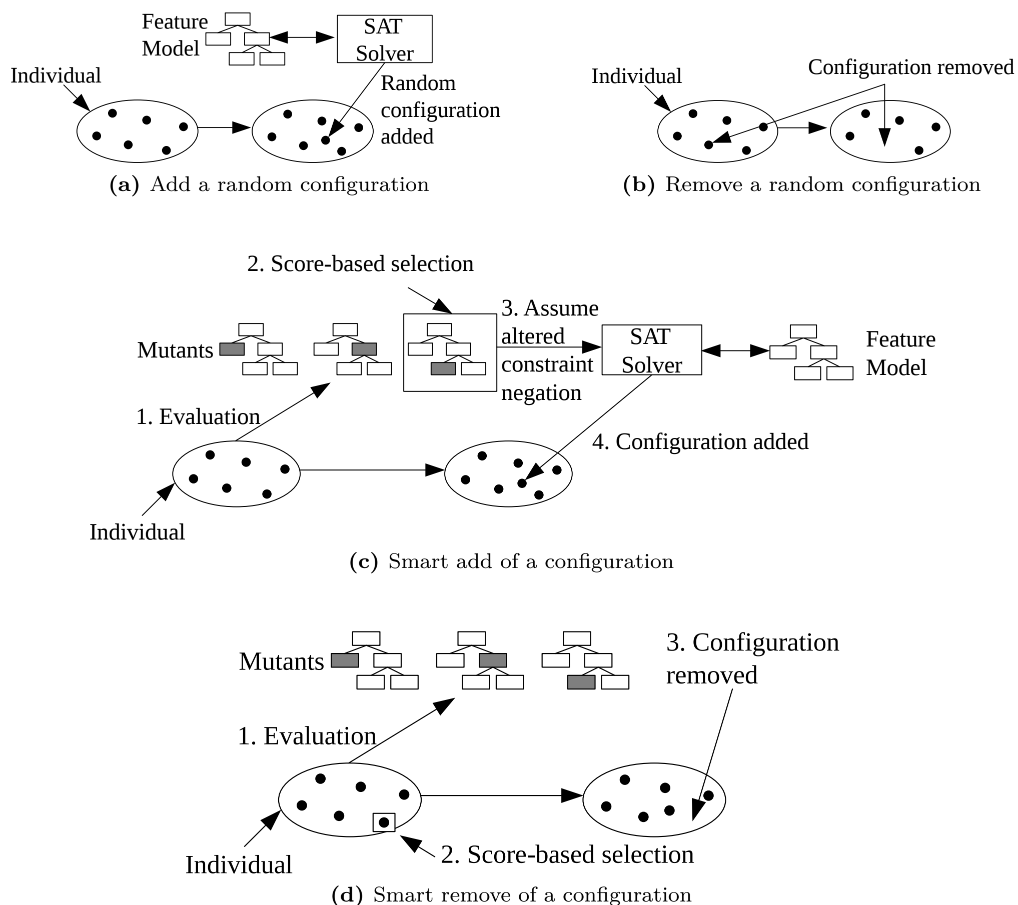
5.3 Experiments

In this section, the proposed SB approach, that we will denote as SB is evaluated on a set of FMs. The objective of these experiments is to answer the two following RQs:

- [RQ1] *Is the proposed approach capable of generating configurations leading to an improved MS?*

- [RQ2] *How does the proposed approach compare with a random one in terms of MS and number of configurations generated?*

The first RQ aims at evaluating whether the MS is increasing over the generations of SB and if at a point it is able to converge. We expect to see the MS increasing over the generations and stabilize at



(a) Add a random configuration

(b) Remove a random configuration

(c) Smart add of a configuration

(d) Smart remove of a configuration

Figure 5.2: The search operators used by the generation approach.

Table 5.1: The feature models used for the experiments.

FM	Features	Constraints	Possible configurations	Mutants
Cellphone	11	22	14	119
Counter Strike	24	35	18,176	208
SPL SimulES, PnP	32	54	73,728	291
DS Sample	41	201	6,912	1,086
Electronic Drum	52	119	331,776	664
Smart Home v2.2	30	82	3.87×10^{9}	434
Video Player	71	99	4.5×10^{13}	582
Model Transformation	88	151	1.65×10^{13}	851
Coche Ecologico	94	191	2.32×10^{7}	1,030
Printers	172	310	1.14×10^{27}	1,829

Table 5.2: Mutation operators used to alter feature models.

Mutation Operator	Action
Literal Omission (LO)	A literal is removed
Literal Negation (LN)	A literal is negated
OR Reference (OR)	An OR operator is replaced by AND

a time. In practice, it means that the approach is capable of improving the solution and reach a good enough MS.

The second question amounts to evaluate how SB compares with a naive approach. Since no other technique exists to perform a mutation-based generation of configurations for SPLs, we compare it to a random one. To this end, two bases of comparison are used. The first one is the evaluation of the MS when generating the same number of configurations with both approaches. The second baseline evaluates the number of configurations required by the random approach to achieve the same level of MS as SB. It is expected that a higher MS than random for the same number of configurations will be observed and we expect a random generation to necessitate more configurations than SB to achieve a given MS.

5.3.1 Approach assessment (research question 1)

5.3.1.1 Setup

SB has been performed 30 times independently per FM with 1,000 generation with an equal probability $p = 0.25$ to apply one of the four operators.

5.3.1.2 Results

The results are recorded in Figure 5.3 and Table 5.3. The figure presents the evolution of the MS averaged on all the FM and all the 30 runs while the table presents detailed results per FM. With respect to Figure 5.3, one can see the ability of the approach to improve the MS over the generations and stabilize around 0.8. With respect to Table 5.3, one may observe that the approach is able to improve the MS for each of the considered FM, with improvements of 68% in average for the DS Sample FM. Besides, there are very small (0.03) or non-existent variations among the different

Table 5.3: Comparison between the initial and final mutation score on the 30 runs for 1,000 generations.

	Generation 1			Generation 1,000		
FM \ MS	min	max	avg	min	max	avg
Cellphone	0.39	0.66	0.5	0.79	0.79	0.79
Counter Strike	0.37	0.56	0.45	0.79	0.79	0.79
SPL SimuleES, PnP	0.42	0.62	0.49	0.7	0.7	0.7
DS Sample	0.17	0.27	0.22	0.9	0.9	0.9
Electronic Drum	0.38	0.56	0.44	0.78	0.78	0.78
Smart Home v2.2	0.45	0.66	0.54	0.89	0.89	0.89
Video Player	0.36	0.55	0.45	0.69	0.72	0.71
Model Transformation	0.41	0.61	0.5	0.86	0.86	0.86
Coche Ecologico	0.44	0.57	0.49	0.8	0.8	0.8
Printers	0.35	0.45	0.41	0.74	0.75	0.75

final MS achieved over the 30 runs, fact demonstrating the ability of SB to reach a good solution at each execution of the approach. Finally, it should be noticed that SB achieves the above-mentioned results using only a small number of generations (1,000 generations). This is an achievement since SB techniques usually require thousands of executions in order to be effective [HM10].

5.3.1.3 Answering research question 1

The results presented in the previous section demonstrate the ability of SB to both improve the MS over the generations and converge towards an acceptable MS. Indeed, some mutants may not be detectable if they are either leading to an invalid formula or an equivalent to the original FM formula (i.e., there is no configuration that cannot satisfy it), thus limiting the maximum score achievable by the approach. In this work, we only focus on the process of generating a configuration suite which maximize the MS. Finally, we observe improvements in the MS of over 60% and a quick convergence, with very small variations between each of the 30 runs, thus giving confidence in the validity of the search approach.

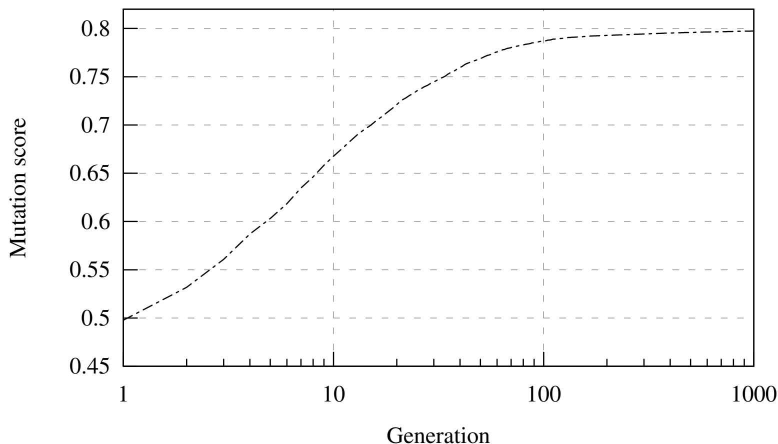


Figure 5.3: Evolution of the mutation score over the 1,000 generations averaged on all the feature models for the 30 runs.

Table 5.4: Comparison between the search-based approach and a random one on the following basis: (a) same number of configurations and (b) same mutation score (MS). Each approach has been performed 30 times independently. #Conf denotes the number of configurations. The execution time is in seconds.

		SB approach			Rand. same #Conf		Rand. same MS	
FM	30 runs	#Conf	MS	Time (s)	MS	Time (s)	#Conf	Time (s)
Cellphone	min	3	0.79	2	0.48	0	4	0
	max	4	0.79	3	0.79	0	42	0
	avg	3.46	0.79	2.66	0.67	0	12.4	0
Counter Strike	min	7	0.8	9	0.68	0	22	0
	max	11	0.8	11	0.75	1	109	2
	avg	9.53	0.8	10.6	0.72	0.16	43.73	0.56
SPL SimulES, PnP	min	3	0.7	11	0.61	0	4	0
	max	5	0.7	13	0.7	1	30	1
	avg	4.36	0.7	11.9	0.66	0.1	9.66	0.16
DS Sample	min	16	0.9	46	0.56	0	32	1
	max	17	0.9	49	0.77	1	114	8
	avg	16.03	0.9	46.8	0.70	0.2	60.26	2.9
Electronic Drum	min	5	0.78	22	0.66	0	9	0
	max	8	0.78	27	0.77	1	29	1
	avg	6.83	0.78	24.8	0.72	0	15.46	0.3
Smart Home v2.2	min	7	0.88	26	0.79	0	13	0
	max	11	0.88	30	0.88	1	43	2
	avg	8.36	0.88	28	0.84	0.1	22.7	0.66
Video Player	min	14	0.69	53	0.62	0	161	19
	max	22	0.72	65	0.65	1	1,000*	532
	avg	18.86	0.71	59	0.64	0.5	518	183
Model Transfo.	min	8	0.86	54	0.77	0	15	0
	max	12	0.86	67	0.85	1	56	4
	avg	9.36	0.86	59.2	0.82	0.2	31.13	1.86
Coche Ecologico	min	11	0.8	75	0.71	0	17	1
	max	14	0.8	89	0.77	1	57	7
	avg	11.76	0.8	80	0.74	0.	31.36	2.9
Printers	min	25	0.74	443	0.67	2	149	110
	max	35	0.75	567	0.72	3	1,000*	4,928
	avg	30	0.75	513	0.70	2.4	481	1,264

*The number of configurations required by random to achieve the same MS as SB has been limited to 1,000.

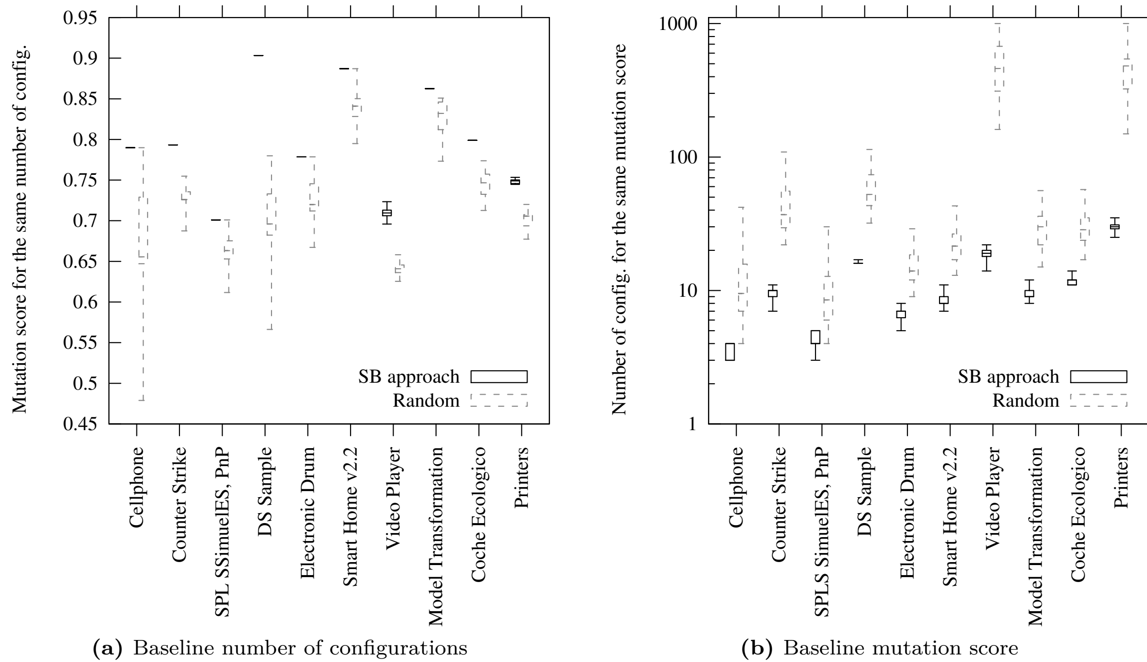
5.3.2 Comparison with random (research question 2)

5.3.2.1 Setup

SB has been performed 30 times independently per FM with 1,000 generation allowed. An equal probability $p = 0.25$ to apply one of the four operators has been set. For each run of SB, a random one has been conducted in order to (a) evaluate the MS achieved when randomly generating the same number of configurations as the number proposed by SB, and (b) evaluate the amount of generated configurations required by the random approach in order to achieved the same MS. In the latter case, a limit of 1,000 configurations has been set.

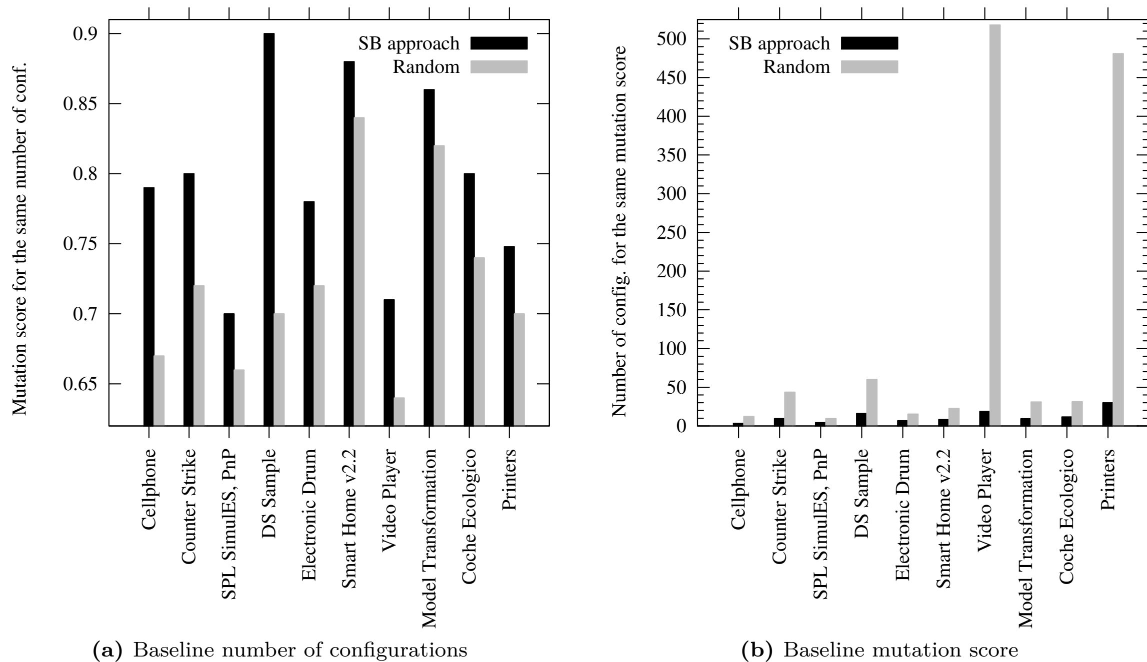
5.3.2.2 Results

The results are recorded in Table 5.4. It presents the minimum, maximum and average number of configurations, MS (MS) achieved and execution time in seconds for the following approaches: SB, random based on the same number of configurations as SB and random based on the same MS as SB.



(a) Baseline number of configurations **(b)** Baseline mutation score

Figure 5.4: Search-based approach VS Random: distribution of the mutation score and number of configurations on the 30 runs.



(a) Baseline number of configurations **(b)** Baseline mutation score

Figure 5.5: Search-based approach VS Random: average values of the mutation score and number of configurations on the 30 runs.

Besides, Figure 5.4 depicts the distribution of the values over the 30 runs and Figure 5.5 presents the average values.

From these results, one can see that SB is quite stable, with small variations in both the MS and number of configurations achieved (5.4b and 5.4a). Compared to random based on the same number of configurations, SB always performs better in terms of MS. For instance, for the DS Sample FM, there is a difference of 0.34 on minimum MS achieved and 0.2 on the average one (Table 5.4). Regarding the comparison based on the MS, the random approach requires much more configurations to achieve the same MS. For instance, with respect to the Video Player FM, the random approach requires in average more than 500 configurations to reach a MS of 0.71 while SB only needs less than 20 (Figure 5.5a). In addition, there were some cases, e.g., the Printers FM where the random approach was not able to achieved the same MS as the one reached by SB, requiring more than 1,000 configurations and more execution time than SB.

5.3.2.3 Answering research question 2

Our results show that SB outperforms the random approach. We observed a difference between random and SB of up to 34 % in favor of SB. Additionally, the random technique requires much more configurations to achieve a given MS. In some cases, it is not even able to terminate, requiring more than 20 times more configurations. This shows the ability of SB to generate configurations while at the same time maximizing the MS that can be achieved.

5.4 Threats to validity

The experiments performed in this chapter are subject to potential threats towards their validity. First, the FMs employed are only a sample and thus the generalization of theses results to all possible FMs is not certain. In particular, using different models might lead to different results. In order to reduce this threat, we selected 10 FMs of different size and complexity. Thus, we tried to use a diversify and representative set of subjects. A second potential threat can be due to the experiments themselves. First, there is a risk that the observed results happened by chance. To reduce this threat, we have repeated the execution of both the proposed approach and the random one 30 times per FM. Doing so allows reducing risks due to random effects. Another threat can be due to the SAT solver used. Indeed, there is a risk that another solver will lead to different results. We choose the PicoSAT solver as it was easy to modify it to produce random solutions. The same threat holds for the mutation operator used. We tried to employ various mutation operators that are relevant for FM formulas. This chapter aims at generating configurations with the aim of detecting mutants. The ability of finding faults is not evaluated. Regarding the MS achieved, it is expected that giving more time to the SB approach will provide better results. Even if small differences are observed in the MS compared to the random approach, this can be in practice leading to finding more faults [PHT14, HPP+14]. Finally, the presented results could be erroneous due to potential bugs within the implementation of the described techniques. To minimize such threats, we divided our implementation into separated modules. We also make publicly available the source code and the data used for the experiments.

5.5 Conclusions

This chapter devised an approach for generating configurations for SPLs based on mutation. The novelty of the proposed technique is the use of mutation of the FM to guide the search, thus focusing on possible faulty implementation of the FM that should be tested. To the authors knowledge, it is the first approach that is performing so. The conducted experiments show the benefit of the approach compared to a random one as it is able to both reduce the configuration suite size while significantly increasing the MS. To enable the reproducibility of our results, our implementation and the FMs used are publicly available at `http://research.henard.net/SPL/SSBSE_2014/`.

Part III

Multi-objective configuration generation

6

A constraint-aware search approach for configuring large software product lines

In the previous part, configuration generation approaches were presented. However, they were aiming at satisfying only one testing objective, combinatorial testing and mutation. This chapter presents a multi-objective configuration generation approach handling multiple testing objectives that target a single configuration.

This chapter is based on the work published in the following paper:

- Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. Combining multi-objective search and constraint solving for configuring large software product lines. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE 2015)*, 2015

Contents

6.1 Introduction **72**
6.2 The existing approaches **73**
6.3 The proposed approaches **73**
6.3.1 Diversity promotion 74
6.3.2 “Smart" operators 74
6.3.3 The SATIBEA approach 75
6.3.4 The Filtered approach 75
6.4 Research questions **76**
6.5 Experimental setup **77**
6.5.1 Subjects 77
6.5.2 Optimization objectives 77
6.5.3 Settings 78
6.5.4 Metrics 78
6.5.5 Statistical analysis and tests 81
6.6 Experimental results **81**
6.6.1 Results 81
6.6.2 Answering research questions 1 & 2 82
6.6.3 Answerwing research question 3 82
6.6.4 Answering research question 4 83
6.7 Discussion **83**
6.7.1 Practical implications 83
6.7.2 Threats to validity 84
6.8 Conclusions **85**

6.1 Introduction

The problem of feature selection was first addressed in 2008 by White et al. [WDS08] who introduced an approach called Filtered Cartesian Flattening to select features from a FM, but this was only able to cater for single optimization objectives. In 2011 Guo et al. [LLWG12, GWW^{+}11] introduced a genetic algorithm for the same problem, demonstrating that it outperformed Filtered Cartesian Flattening on synthetically generated SPLs, but did not present results for any real-world SPLs.

These previous approaches were all single objective approaches. Therefore they could not construct software products from SPLs for which multiple (perhaps conflicting and competing) objectives needed to be optimized. Sadly, such single objective solutions are unsuited to most real-world SPL feature selection problems (which are multi-objective). However, in 2011, Wu et al. [WTKC11] introduced a multi-objective optimization formulation that was evaluated on a Mail Server System case study.

In 2013 Sayyad et al. provided a detailed investigation of the multi-objective SPL feature selection problem in four related papers [SGPMA13, SIMA13a, SMA13, SIMA13b] that collectively established the current state-of-the-art. Their first paper [SMA13] demonstrated that search-based optimization can be used to find products that optimize multiple objectives. They evaluated on real-world SPLs, replicating their results [SGPMA13], and reporting on parameter tuning effects [SIMA13a]. Finally, Sayyad et al. introduced additional heuristics to improve the scalability of their approach [SIMA13b], which is an important consideration for SPL optimization, since SPLs can be very large.

None of these previous approaches to SPL feature selection have included any explicit technique to handle constraints, leaving open the question of how best to optimize SPL feature selection in the presence of constraints. This is an important open question because most real-world SPLs are highly constrained [LP07], and solutions that fail to respect such constraints are likely to be rejected by both developers and their users.

Indeed, many constraint-violating solutions will prove to be simply *unbuildable*; constraints often determine whether or not a product can be feasibly constructed. Furthermore, this chapter shows that concentrating on constraint-respecting solutions also allows the search to find software products that significantly outperform the state-of-the-art.

Contributions of this chapter. We introduce SATIBEA, a search-based SPL configuration generation algorithm, augmented by constraint solving and two smart search operators. SATIBEA guides the automated search to constraint-respecting solutions that maximise multiple objectives in reasonable time. Our empirical study, which include the largest yet reported SPL, demonstrates that SATIBEA is a scalable and significant improvement over the current state-of-the-art.

The primary contributions of the chapter can be summarised as follows:

1. We introduce SATIBEA, a new algorithm for SPL selection and evaluate it on 5 real-world SPLs, ranging from 1,244 to 6,888 features with respect to 3 quality indicators and two diversity measures. We perform 30 independent executions to support inferential statistical testing for significance and assessment of effect size.

2. We show that SATIBEA significantly outperforms the current state-of-the-art (with maximal effect size) according to all 3 solution quality indicators and for all 5 SPLs.

3. We demonstrate the importance of augmenting search with constraint solving in such constrained spaces as SPLs: We present results that show that our simple constraint solving approach alone can also significantly outperform the state-of-the-art with maximal effect size with respect to all 3 solution quality indicators in 3 SPLs including the largest one, Linux.

4. We demonstrate the added value of our combined approach with smart operators over constraint solving alone. Over the 15 comparisons (5 SPLs, each with 3 quality indicators) we find that SATIBEA significantly outperforms constraint solving alone in 13, and with maximal effect size in 11.
5. We demonstrate SATIBEA's scalability. Scalability is a known and important issue for both SPLs [SIMA13b, LP07, HPP$^+$14] and search-based software engineering [HMZ12].

The remainder of the chapter is organized as follows: Sections 6.2 present the existing work and Section 6.3 details the proposed approaches. The studied RQs and the experimental setup are detailed in Sections 6.4 and 6.5. Experimental results are presented and discussed in Sections 6.6 and 6.7. Finally, Section 6.8 concludes the chapter.

6.2 The existing approaches

The IBEA [ZK04] is an evolutionary MOO technique using quality indicators to guide the search towards the optimal solutions. IBEA has the ability to exploit user preferences. The advantages of this algorithm over other search techniques for the SPL configuration problem have been shown in [SMA13, SIMA13b].

Sayyad *et al.* [SMA13] proposed setting the number of constraints that are violated as a minimization objective within the search process in order to deal with the SPL constraints. The user preferences are also modeled as additional optimization objectives. The approach applies the standard mutation, i.e., flipping bits of the offspring with a specific probability, and crossover operators. Their results provide evidence that this practice can lead to invalid and marginally invalid configurations. They also suggest that IBEA is capable of providing a wide range of valid configurations that exploit and optimize user preferences.

The results of Sayyad *et al.* were reinforced by the study of Olaechea *et al.* [ORGC14, Ola13] who demonstrated, on small models, that IBEA is capable of finding the optimal solutions. Olaechea *et al.* also showed that is feasible to compute exact solutions when considering models with fewer than 45 features. This bound indicates the need for approximation algorithms, such as IBEA, for the cases of larger models.

Empirical evidence has been provided to show that by enhancing the initial population of the algorithm with one valid configuration, called *seed*, IBEA is capable of scaling on very large FMs has also been provided [SIMA13b]. According to the studies of Sayyad *et al.*, one seed that is rich, i.e., one configuration with many features selected, is adequate for improving the search process and more effective than using many seeds. We only consider large SPLs and thus, we compare with this approach which forms the current state-of-the-art. In the rest of the chapter we refer to it as the *state-of-the-art* or as the *IBEA* approach.

6.3 The proposed approaches

One of the most challenging SPL optimization tasks is the automatic generation of valid configurations. The current state-of-the-art uses IBEA to search and find valid solutions. An alternative to search would involve the use of a SAT solver [HPP$^+$14]. In this case, a valid configuration is a satisfiable "model" found by the solver. To this end, one might attempt to enumerate all the valid solutions of a model and select those that are optimal with respect to the other objectives. However, the large

number of valid configurations makes this simplistic approach infeasible [Ola13]. As a result, some form of search-based technique is needed.

To effectively perform search, we seek to combine the benefits of both constraint solving and searching in a complementary way. The question this raises is how best to perform such a combination. To achieve this, two key aspects are considered: diversity promotion and search using smart operators. These aspects are taken into account in our approach called *SATIBEA*. We also define a "filtered" technique which only bestows the diversity promotion. Doing so allows to empirically assess its contribution to SATIBEA success in isolation.

6.3.1 Diversity promotion

We wish to promote maximal diversity of SAT solutions in a cheap way. We do this by randomly permuting the parameters that control the search for constraint-satisfying solutions processed by the SAT solver. More specifically, there are three different SAT parameters that we permute:

1. **Constraint order**. This is the order in which the constraints are considered.
2. **Literal order**. This is the order in which the literals of each constraint are ordered.
3. **Phase selection**. This is the order $\{true, false\}$ in which assignments to variables are instantiated.

By randomly permuting these three parameters at each iteration of the SAT execution, we increase the diversity of solutions found. To empirically assess the degree of Diversity Promotion (DP) this creates, we use a dissimilarity metric, as it is defined in [HPP+14]. Based on the Jaccard distance, this metric captures degrees of difference between the selected and unselected features of two configurations. The metric takes values between 0 and 1. A value of 1 signifies that the two configurations differ completely, while, 0 signifies that the two considered configuration are the same.

Table 6.1 records the results of the above-mentioned dissimilarity metric for solutions produced, on a set of subject models, with and without DP. These subjects are introduced in Section 6.5.1. Specifically, Table 6.1 records a) the average dissimilarity between two configurations that are consecutively generated by calling the solver 1,000 times, b) the set dissimilarity, i.e., the dissimilarity between any two configurations from a set of 1,000 configurations produced by the solver and c) the percentage increase in the diversity of the configurations as measured by the a) and b) cases. The dissimilarity between two consecutive configurations C_i, C_j is measured by $d(C_i, C_j) = \frac{|C_i \cup C_j| - |C_i \cap C_j|}{|C_i \cup C_j|}$. The dissimilarity of a set of n configurations is measured by $D(C_1, ..., C_n) = \frac{1}{\binom{n}{2}} \sum_{j>i}^{n} d(C_i, C_j)$. Finally, the increase in the diversity is calculated as follows: increase $= \frac{\text{(with DP}-\text{without DP)}}{\text{without DP}} \times 100$. As it can be seen in Table 6.1, the permutation of the SAT parameters allows the diversity of the solutions to increase by 2,768% in the worst case for the consecutive calls to the solver. For the set of configurations, the diversity increase was 161% in the worst case and more than 39,000% in the best case.

6.3.2 "Smart" operators

We introduce two operators that are "smart" in the sense that they are constraint-aware and using diversity promotion.

6.3.2.1 Smart mutation

This mutation operates by finding the features that are not involved in the violations of constraints. It keeps their values and asks the solver to find a solution for the rest by assuming the values of the rest of the features. Consider an FM with 5 features and 3 constraints: $FM = (f_1 \vee f_5) \wedge (f_2 \vee f_3) \wedge (f_2 \vee f_5)$. The configuration $C = \{\overline{f_1}, \overline{f_2}, f_3, f_4, \overline{f_5}\}$ is invalid because the two constraints $(f_1 \vee f_5)$ and $(f_2 \vee f_5)$, which involve the features f_1, f_2 and f_5, are violated. We remove the assignment of these features and make C partially valid, i.e., $C : C_{\text{partial}} = \{_, _, f_3, f_4, _\}$. This partial configuration is given to the SAT solver, which will complete it, to return a valid configuration. For instance, it can return the following configuration $C' = \{f_1, f_2, f_3, f_4, \overline{f_5}\}$. As a result, C has been mutated into C'.

6.3.2.2 Smart replacement

This operator randomly picks a configuration from the solutions and replaces it with a new valid one, improving the quality and diversity of the solutions.

6.3.3 The SATIBEA approach

SATIBEA augments IBEA [ZK04] with the smart operators. Diversity promotion is used in the optimization process through these two operators. SATIBEA also employs a form of memory by keeping track of all the valid configurations produced by the algorithm. Based on these solutions, we compute the Pareto front. Thus, the population is evolved via the four following operators:

1. **Mutation**. This is the standard bit-flip operator of IBEA. It iterates over the bits, i.e., the feature options, of the offspring, i.e., the configuration, and flips them with a specific probability.

2. **Crossover**. This is the “standard” single-point crossover operator of IBEA. It combines two solutions, i.e., configurations, by replacing the bits of the first one, from the beginning of the offspring up to the crossover point, with those of the second one.

3. **Smart Mutation**, as described in Section 6.3.2.1.

4. **Smart Replacement**, as introduced in Section 6.3.2.2.

6.3.4 The Filtered approach

To investigate the contribution of the diversity promotion to SATIBEA’s performance, we also define a simple algorithm that simply randomly samples over diversity promoted SAT solutions. We refer to this approach as the *Filtered* one.

Table 6.1: Dissimilarity with and without Diversity Promotion (DP) on 1,000 configurations per feature model.

	Consecutive configurations			**Set of configurations**		
	without DP	with DP	increase	without DP	with DP	increase
Linux	0.0004	0.5934	148,250%	0.0015	0.5937	39,487%
uClinux	0.0036	0.2807	7,697%	0.1080	0.2814	161%
Fiasco	0.0066	0.1892	2,768%	0.0436	0.1869	329%
FreeBSD	0.0022	0.5891	26,677%	0.0074	0.5897	7,991%
eCos	0.0046	0.5429	11,702%	0.0304	0.5426	1,685%

6.4 Research questions

We first empirically evaluate SATIBEA against the current state-of-the-art [SMA13, SIMA13b]. This is a natural first RQ, since there is no point in evaluating further if our new algorithm cannot convincingly outperform the state-of-the-art.

- [RQ1] *How does the SATIBEA compare with the current state-of-the-art?*

Since the results of RQ1 indicate that SATIBEA does, indeed, convincingly outperform the state-of-the-art, we turn to the question of examining why. Naturally, since one of our primary novelties lies in the incorporation of SAT solving into the search for constraint-respecting solutions, we next investigate and report on the effectiveness of SAT solving alone. How well would SAT solving perform against the current state-of-the-art on its own? This motivates RQ2:

- [RQ2] *How well does the state-of-the-art perform against constraint solving alone (randomly selected solutions filtered by SAT, i.e., the Filtered approach)?*

Perhaps surprisingly, we found that the Filtered approach outperforms the state-of-the-art. This provides compelling evidence that constraint solving does have an important role to play in the search for optimized products, automatically configured from SPLs. However, it also raises a further question: does SATIBEA significantly outperform constraint solving alone? If the answer is 'no', then all the value in our new SATIBEA approach derives from our incorporation of constraint solving, with search-based optimization and our smart mutation operators offering little added value. In order to check that this is *not* the case, we investigate RQ3 below:

- [RQ3] *How well does SATIBEA perform against constraint solving alone (randomly selected solutions filtered by SAT, i.e., the Filtered approach)?*

At this point in our study we will have considered whether our new algorithm SATIBEA outperforms the state-of-the-art (RQ1), whether constraint solving plays an important role in its performance (RQ2) and whether SATIBEA adds value to the search for constraint-respecting optimized software products over-and-above pure constraint solving alone (RQ3). Our final question concerns the execution time required to achieve these results. Even if SATIBEA convincingly outperforms all alternatives, this will be of little consequence if it does not scale well to the challenges of very large SPLs involving billions of possible configurations over thousands of features. We therefore conclude our study by reporting on the time taken to complete the execution of SATIBEA on the largest SPL for which results have been reported in the literature to date.

- [RQ4] *What is the execution time required to find constraint-respecting optimized software products from the largest SPL hitherto considered in the literature?*

Table 6.2: Feature models used in the empirical study.

Feature model	Version	Features *(mandatory)*	Constraints
Linux [SLB+11]	2.6.28.6	6,888 *(58)*	343,944
uClinux [BSL+12]	20100825	1,850 *(7)*	2,468
Fiasco [BSL+12]	2011081207	1,638 *(49)*	5,228
FreeBSD [SLB+11]	8.0.0	1,396 *(3)*	62,183
eCos [SLB+11, BSL+10]	3.0	1,244 *(0)*	3,146

6.5 Experimental setup

This section presents the settings of the conducted experiments. Specifically, it describes the subjects, the optimization objectives and the employed metrics.

6.5.1 Subjects

The study uses 5 FMs taken from the Linux Variability Analysis Tools (LVAT) repository[i]. The characteristics of the FMs are described in Table 6.2. For each of them, it presents the version used, and the number of features and constraints it contains. Following the evaluation approach used by Sayyad *et al.* [SMA13, SIMA13b], each feature of each FM has been augmented with 3 attributes: *cost*, *used before* and *defects*. The values for these attributes have been set arbitrarily with a uniform distribution: *cost* takes real values between 5.0 and 15.0, *used before* takes Boolean values and *defects* takes integer values between 0 and 10. The following dependency among these attributes is used: if (not *used before*) then *defects* = 0.

6.5.2 Optimization objectives

In this study, we are measuring the following 5 objectives:

1. *Correctness.* We seek to minimize the constraints of the FM that are violated by a configuration.
2. *Richness of features.* We seek to minimize the number of deselected features in a configuration.
3. *Features that were used before.* We seek to minimize the features that were not used before, i.e., minimize the number of “false” for this attribute.
4. *Known defects.* We seek to minimize the number of known defects in a configuration.
5. *Cost.* We seek to minimize the cost of a configuration.

In practice, based on the needs and the historical data of engineers, other objectives can be also used. We selected these five objectives to ensure identical settings as those reported for the state-of-the-art [SIMA13b].

[i] `http://code.google.com/p/linux-variability-analysis-tools`

Table 6.3: Selected features in the seeds used by IBEA.

Feature Model	Selected features	Selected features in [SIMA13b]
Linux	6,265	5,704
uClinux	613	455
Fiasco	338	575
FreeBSD	1,088	946
eCos	1,148	967

6.5.3 Settings

All the experiments were performed on a Quad Core@2.40 GHz with 24GB of RAM. To enable a fair comparison with the state-of-the-art, we used exactly the same settings as the ones of Sayyad *et al.* [SIMA13b]. These settings are: population size *300*, archive size *300*, crossover rate *0.05* and *0.001* mutation probability. The mutation probability refers to the probability that an optional feature of the model will be flipped. Regarding the configurations, we systematically set mandatory features (features that have to be present in any configuration) as selected and dead ones (features that cannot be part of any configuration) as unselected in the initial population of IBEA. We also prevented IBEA from flipping these features during the mutation process. Flipping these features always leads to invalid configurations. Thus, this practice helps IBEA to find valid configurations. The same evaluations settings were undertaken in the study of Sayyad *et al.* [SIMA13b].

We carefully followed all the recommendations of Sayyad *et al.* in our experiments. Unfortunately, it is impossible to produce the same seeds. Since the work of Sayyad *et al.* [SIMA13b] is not currently accompanied by any data or implementation, we simply followed the guidelines they give in their paper. Therefore, we have produced seeds using the solver by maximizing the number of selected features. This is done by setting the SAT parameter “phase selection" to assign *true* to the variables. Note that this parameter was also used for diversity promotion (see Section 6.3.1). We thus produce one “rich" seed per model, as suggested by Sayyad *et al.* [SIMA13b]. Table 6.3 describes the number of features selected in each seed.

Similarly, the settings for SATIBEA are the same as for IBEA, i.e., population size *300*, archive size *300*, crossover rate *0.05*. The probability to use the standard mutation of IBEA (bitflip), which mutates a chromosome is set to 0.98. The probability to flip a feature is set to *0.001* per feature. The probabilities of mutating using the smart mutation and the smart replacement is 0.01 for both cases. We employed the Sat4j SAT solver [BP10] and used the jMetal framework [DN11] for the implementation of IBEA and for the quality and diversity metrics (see Section 6.5.4). We independently applied each approach 30 times per FM with 30 minutes of execution time for each algorithm. Invalid configurations were discarded for all the studied techniques. Recall that invalid configurations are useless in practice.

6.5.4 Metrics

To evaluate the studied approaches, we follow two directions: 1) we measure the proximity of the solutions found from the optimal ones, i.e., their quality, and 2) we evaluate the diversity of the solutions. Note that diversity is only useful when there is quality: a single diamond is preferable to an arbitrary number of diverse glass fragments. In other words, it is useless to have diverse solutions that are all dominated by a single one. Therefore, the diversity metrics should be considered only when comparing solutions of similar quality.

Since the global optimum cannot be known in all cases (as with all NP-hard problems), a reference front is used in evaluation. It consists of the best solutions found by all the studied approaches and it is defined as follows: Given n Pareto fronts $A_1, ..., A_n$, and if $1 \leq j \leq m \leq n$, the reference front A_{ref} is defined as: $A_{\text{ref}} = \{x_1, ..., x_m \mid (\forall x_j \in A_{\text{ref}})(\nexists x' \in \bigcup_{i=1}^{n} A_i)(x' \succ x_j)\}$. It should be noted that $A_{\text{ref}} \subseteq A_i \cup ... \cup A_n$.

6.5.4.1 Quality metrics

These metrics ensure that we find high quality solutions. Following the evaluation approach suggested by Knowles *et al.* [KTZ06], we use three metrics to evaluate the quality of the configurations of the Pareto front: *Hypervolume*, *Epsilon* and *Inverted Generational Distance*.

Hypervolume (HV) This metric represents the volume of the objective space that is dominated by the Pareto front A. It evaluates how well a Pareto front fulfills the optimization objectives. It is written HV and defined in [BFN08] as follows: $\text{HV}(A) = \lambda \left(\bigcup_{x \in A} [F_1(x), r_1] \times \cdots \times [F_k(x), r_k] \right)$, where $\lambda(S)$ is the Lebesgue measure [Haw01] of a set S, k is the number of objectives, $r = [r_1, ..., r_k]$ is the reference point and $[F_1(x), r_1] \times \cdots \times [F_k(x), r_k]$ is the k-dimensional hypercuboid consisting of all the points dominated by the point x. The reference point is the maximum value that belongs to the reference front. A higher HV denotes a better Pareto front.

Epsilon (ε) This metric measures the shortest distance that is required to transform every solution in a Pareto front A to dominate the reference front [KTZ06]. If $x = [x^1, ..., x^k]^T \in \mathbb{R}_k^+$ is a solution, it is defined as [ZTL+03]: $\varepsilon(A, A_{\text{ref}}) = \inf_{x \in \mathbb{R}} \{\forall x' \in A_{\text{ref}} \, \exists x \in A \mid x \succeq_\varepsilon x'\}$, where $x \succeq_\varepsilon x'$ if and only if $\forall 1 \leq i \leq k : x^i \leq \varepsilon \cdot {x'}^i$. This indicator denotes how close A is to the reference front and thus, lower values are preferable.

Inverted generational distance (IGD) This metric is the average distance from the solutions belonging to the reference front to the closest solution in a Pareto front A [DAVV98]. IGD is defined as follows: $\text{IGD}(A, A_{\text{ref}}) = \frac{\sum_{x' \in A_{\text{ref}}} d(x', A)}{\text{PFS}(A_{\text{ref}})}$, where $d(x', A)$ is the minimum Euclidean distance between x' and the other points in A and PFS is the Pareto Front Size (see Section 6.5.4.2). For ε, the lower the value of IGD, the closer A is to the reference Pareto front.

6.5.4.2 Diversity metrics

These metrics ensure that the decision maker has a variety of solutions to choose. We use two diversity metrics: the *Pareto front size* and the *Spread* of the solutions in the explored space.

Pareto front size (PFS) This metric is the number of solutions in a Pareto front A. It is calculated as the cardinality of the Pareto front set, i.e., $\text{PFS}(A) = |A|$. A higher Pareto front size is preferred since more options are given to the user. However, this is only important when high quality is preserved.

Table 6.4: State-of-the-art VS the proposed approaches: comparison in terms of quality metrics, i.e., hypervolume (**HV**), epsilon (ε) and inverted generational distance (**IGD**), and diversity metrics, i.e., Pareto front size (**PFS**), spread (**S**) on 30 independent runs per approach. Higher values are preferred for **HV**, **PFS** and **S**. Lower values are preferred for ε and **IGD**.

			IBEA (I)		**Filtered (F)**		**SATIBEA (SI)**		**SI VS I**		**F VS I**		**SI VS F**	
			median	avg	median	avg	median	avg	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$
Linux	Quality	**HV**	7.75E-7	1.00E-6	0.1133	0.119	0.1624	0.1627	3.02E-11	1	3.02E-11	1	4.62E-10	0.97
		ε	0.9084	0.9082	0.1623	0.1616	0.0733	0.0721	2.96E-11	1	2.97E-11	1	3.02E-11	1
		IGD	0.0177	0.0177	0.0014	0.0014	0.0003	0.0003	3.02E-11	1	3.02E-11	1	3.02E-11	1
	Diversity	**PFS**	7	7.433	117	116.7	111.5	110.8	2.54E-11	1	2.56E-11	1	4.43E-07	0.11
		S	0.9988	0.9988	0.8679	0.8844	0.9004	0.8997	1.11E-06	0.14	6.77E-11	0.21	0.40	0.56
uClinux	Quality	**HV**	0.1956	0.1959	0.1300	0.1297	0.3030	0.3032	3.02E-11	1	3.02E-11	0	3.02E-11	1
		ε	0.1029	0.1018	0.0783	0.0783	0.0107	0.0101	2.87E-11	1	2.38E-11	1	2.30E-11	1
		IGD	0.0013	0.0013	0.0016	0.0016	0.0001	0.0001	2.97E-11	1	2.60E-10	1	2.97E-13	1
	Diversity	**PFS**	106	106.9	982.5	981.9	2,934	2,941	2.95E-11	1	2.95E-11	1	3.02E-11	1
		S	0.5809	0.5804	0.5125	0.5138	0.3610	0.3574	3.02E-11	0	1.20E-08	0.07	3.02E-11	0
Fiasco	Quality	**HV**	0.0238	0.0226	0.2879	0.2877	0.2894	0.2897	3.02E-11	1	3.02E-11	1	2.92E-09	0.95
		ε	0.0833	0.0842	0.0036	0.0036	0.0036	0.0035	1.01E-11	1	1.10E-11	1	0.67	0.45
		IGD	0.0064	0.0065	0.0002	0.0002	0.0002	0.0002	3.02E-11	1	3.02E-11	1	0.49	0.56
	Diversity	**PFS**	10	9.933	2,232	2,231	1,928	1,920	2.74E-11	1	2.74E-11	1	3.01E-11	0
		S	0.9721	0.9282	0.3039	0.3037	0.2959	0.2953	3.02E-11	0	3.02E-11	0	4.08E-05	0.20
FreeBSD	Quality	**HV**	0	0.0257	0.1323	0.1328	0.2485	0.2488	8.38E-10	1	8.39E-10	0.95	3.02E-11	1
		ε	1	0.7926	0.1861	0.1860	0.0953	0.0953	1.61E-11	1	1.61E-11	1	2.98E-11	1
		IGD	1	0.6022	0.0013	0.0013	0.0002	0.0002	1.61E-11	1	1.60E-11	1	3.02E-11	1
	Diversity	**PFS**	0	5.333	476.5	475.7	1,386	1,383	1.62E-11	1	1.60E-11	1	3.00E-11	1
		S	0	0.3990	0.5959	0.5949	0.7197	0.7185	3.02E-11	0.62	1.60E-11	0.62	3.02E-11	1
eCos	Quality	**HV**	0.0399	0.0462	0.2591	0.2591	0.2876	0.2876	3.02E-11	1	3.02E-11	1	3.02E-11	1
		ε	0.5974	0.5906	0.0975	0.0975	0.0382	0.0386	3.02E-11	1	2.53E-11	1	2.53E-11	1
		IGD	0.0013	0.0013	0.0002	0.0002	5.80E-05	5.77E-05	3.02E-11	1	3.02E-11	1	3.02E-11	1
	Diversity	**PFS**	50	55.17	2,886	2,881	14,421	14,064	3.00E-11	1	2.99E-11	1	3.02E-11	1
		S	0.9386	0.9414	0.4551	0.4548	0.5368	0.5364	3.02E-11	0	3.02E-11	0	3.02E-11	1

Spread (S) The spread measure defines the extent of spread in the solutions of the Pareto front A. It is defined in [DPAM02] as follows: $\mathrm{S}(A) = \frac{d_f + d_l + \sum_{i=1}^{\mathrm{PFS}(A)-1} |d_i - \overline{d}|}{d_f + d_l + (\mathrm{PFS}(A)-1)\overline{d}}$, where d_i is the Euclidean distance between consecutive solutions of A, $\overline{d}$ is the average of the d_i's and d_f and d_l are the Euclidean distance between the extreme solutions and the boundary solutions of A. A higher spread denotes a better Pareto front since it reflects more diverse solutions, i.e., distributed among all the optimization objectives.

6.5.5 Statistical analysis and tests

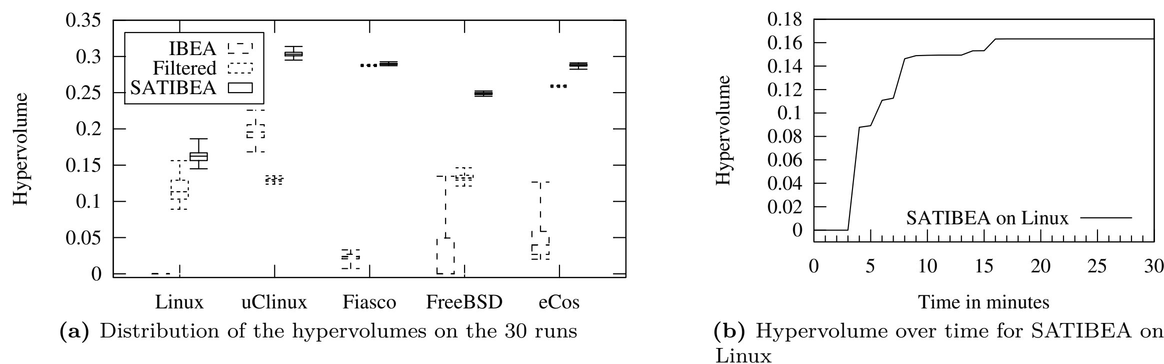
To check the statistical significance of the differences between the algorithms, we performed a statistical test using the Mann–Whitney *U* test (two-tailed) at a 5% significance level. Furthermore, to reduce the threats of having type I errors in the cases of multiple comparisons, i.e., incorrect rejection of a true null hypothesis, we also consider the standard Bonferroni adjustment [AB11]. This is a conservative but safe adjustment because it reduces the chances of type I errors. Following the advice of Arcuri and Briand and Wohlin *et al.* [AB11, WRH+00], we also report the non-parametric effect size measure, $\hat{A}_{12}$, introduced by Vargha and Delaney [VD00]. It measures the extent to which the first algorithm outperforms the second one. According to Vargha and Delaney [VD00], the differences between populations are considered as small, medium and large when $\hat{A}_{12}$ is over 0.56, 0.64, and 0.71, respectively.

6.6 Experimental results

The results for each approach are analyzed in Section 6.6.1. Sections 6.6.2 and 6.6.3 discuss the RQ1-RQ3. Finally, Section 6.6.4 presents results regarding the execution time of SATIBEA on the largest SPL of the literature.

6.6.1 Results

This section presents the result of the approaches when applied to the five models. These results are recorded in Table 6.4. This table is composed of two parts. The columns *IBEA (I)*, *Filtered (F)* and *SATIBEA (SI)*, records the measured details about each approach. In particular, it records, for 30 executions the median and average (column avg) values of the measured metrics. The second part, i.e., columns *SI VS I*, *F VS I*, and *SI VS F*, records the results of the statistical analysis results, i.e., the *p*-values and the effect sizes $\hat{A}_{12}$. The rows of the table record the results per examined model, hypervolume measure (rows HV), Epsilon (rows ε), Inverted Generational Distance (rows IGD), Pareto front size (rows PFS) and spread metric (rows S) for the the 30 runs per approach. In addition, Figure 6.1a shows the distribution of the HVs on the 30 runs for all the models and the evolution of the HV over time for Linux is depicted in Figure 6.1b.



(a) Distribution of the hypervolumes on the 30 runs

(b) Hypervolume over time for SATIBEA on Linux

Figure 6.1: Distribution and evolution of the hypervolumes.

6.6.2 Answering research questions 1 & 2

Our results indicate that SATIBEA outperforms the current state-of-the-art (IBEA). The statistical analysis results, with respect to the quality metrics, column *SI VS I*, suggest that all the differences are significant with maximal effect size ($\hat{A}_{12} = 1.0$). The resulting p-value are so small that nothing changes when applying the Bonferroni correction. Regarding the diversity metrics, SATIBEA provides better results with respect to PFS but worse with respect to S. However, S is a diversity metric which is only important when there is quality in the solutions found, e.g., HV.

Additionally, these results reveal that SATIBEA is much better when it is applied on the two most heavily constrained models, i.e., Linux and FreeBSD. In the median case of Linux (Figure 6.1), SATIBEA produces configurations that cover approximately 209,548 times more wider space, i.e., hypervolume, than IBEA. In the median case of FreeBSD, IBEA failed to find even one solution that is valid. Furthermore, the solutions found by SATIBEA do not only provide more options, as shown by the PFS values, than the state-of-the-art but they are also more stable, as shown in Figure 6.1a. All these results indicate the superiority of our method.

Our results also indicate that the Filtered approach is better than the state-of-the-art. The statistical analysis results, with respect to the quality metrics and column *F VS I*, suggest that in all but the uClinux model, the differences are significant. The differences have high effect size ($\hat{A}_{12} \geq 0.95$) on all the four models it wins. Also, the statistical results do not change by applying the Bonferroni correction. Similarly to SATIBEA, in the cases of diversity, the Filtered provides better results with respect to PFS, but worse with respect to S. However, as already mentioned, this does not indicate that IBEA is better.

Conclusively, both proposed approaches are better than the current state-of-the-art. Noticeable is the fact that SATIBEA wins the current state-of-the-art in all the employed quality metrics with maximal effect size ($\hat{A}_{12} = 1.0$). In addition, when a more heavily constrained is considered, SATIBEA performs much better than the state-of-the-art.

6.6.3 Answerwing research question 3

Our results indicate that SATIBEA wins the Filtered method in all the models according to the HV metric. For instance, for FreeBSD, the median HV achieved by SATIBEA is almost twice the one of Filtered, i.e., ≈ 0.25 VS ≈ 0.13. These results are also statistically significant, both with and without Bonferroni correction, with a relatively high effect size (above $\hat{A}_{12} = 0.95$) on all the case. According to the Epsilon and IGD metrics SATIBEA wins in all the models, with statistical significance, except from the Fiasco where they are approximately equal. Regarding the effect sizes of the diversity measures, the two approaches are comparable with SATIBEA having a slight advantage. With respect to S, SATIBEA has a big difference in two cases, a medium difference in one and it looses or it is equal in two cases. With respect to PFS, it wins in three cases and looses in two. Therefore, since diversity is not so important if we do not have quality, overall, our results demonstrate that SATIBEA is the clear winner. It provides the best results, statistically significant, and can handle effectively heavily constrained FMs.

Conclusively, answering RQ3, our results suggest that SATIBEA is able to outperform the Filtered approach, with HV values ranging from 0.16 to 0.30.

6.6.4 Answering research question 4

Our results indicate that the HV of the solutions achieved by SATIBEA converges markedly the first 15 minutes (Figure 6.1b). After this point, the HV increases very slowly, suggesting that SATIBEA stabilised on its ultimate solution in 15 minutes. This is an important finding since Linux is the largest available SPL hitherto reported upon in the literature. Both the smart replacement and the smart mutation operators which use the solver take less than six seconds. Thus, they help the fast convergence of the search process.

6.7 Discussion

This section discusses practical implications and threats to the validity of the findings reported in the present work.

6.7.1 Practical implications

In the SPL context, the major challenge is the production of valid configurations. It is clear that until all constraints are satisfied, the configuration is invalid. In other words, an invalid product configuration is totally useless from practical perspective. Therefore, the effort put by the search approach in optimizing the other objectives is wasted when the resulting configuration is invalid. To investigate this, we analyze the results of the Linux FM. Specifically, we group the Pareto front solutions of IBEA according to the number of violated constraints. We visualize this situation by computing the hypervolume values when the algorithm minimizes the violated constraints. Figure 6.2 depicts the hypervolume achieved by the Pareto front solutions according to each number of violated constraints. We can observe that as the number of violated constraints decreases, the hypervolume also decreases. These results show that the constraints hamper the search process. Indeed, the graph clearly suggests a decreasing trend. This is formally confirmed by a linear regression which bestows a relationship of the form $f(x) = (5.67 \times 10^{-5})x - 0.003$. This fit is good given its coefficient of determination R^2 of 0.95. In other words, our linear f explains 95% of the recorded values. Finally, it is clear that when no constraint is violated, the hypervolume is almost 0. Since the hypervolume represents how well the objectives are optimized, it shows that IBEA fails to fulfill the 4 other objectives when dealing with valid configurations. As a result, the optimization of the other objectives is limited by the minimization of the violated constraints. This explains why IBEA performs poorly compared to the Filtered approach. These results introduce the need for handling constraints independently of the search. Thus, hybrid methods like SATIBEA are the key to success.

Our evaluation focuses on large SPLs because they are typically used in industry [SIMA13b, LP07, WDS09] and they motivate the need for automation. Generally, our results show that when the number of constraints increases, the difficulty faced by the search approaches also increases. Fortunately, our results reveal that a higher number of constraints implies a higher gap between the effectiveness of the proposed and the current state-of-the-art approaches. Additionally, it should be noted that SATIBEA has an additional benefit over the state-of-the-art: it does not require any seeds. Thus, it avoids the necessary off-line computation of the seeds, which according to Sayyad *et al.* [SMA13] consumed approximately 3 hours.

Apart from optimizing the objectives, the proposed approach can have additional applications. An interesting one is to help engineers into correct and maintain FMs. To achieve this, engineers can select multiple FM variants that represent the potential problems or changes in the original FM. Then, valid configurations (with respect to the FM variants) can be selected and evaluated towards the original FM. Such an approach can be automated as proposed by Henard *et al.* [HPP+13a]. The important step here is the selection of valid configurations from the erroneous FM variants that will reflect both the potential problems and the targeted changes of the original FM. One could argue that this can be achieved with invalid configurations of the original FM. However, it is unclear how to produce invalid configurations that are both relevant and helpful in correcting the model.

Finally, we note the importance of the various quality metrics. The hypervolume metric represents the extent in the optimization of user objectives. Any improvements of this metric yields a strictly better quality value [ZBT07]. In other words, a very small change in the hypervolume implies a relatively big impact in practice. Regarding the spread diversity measure, spread configurations represents solutions that are diverse in the space of the objectives, i.e., which achieve different trade-offs. Concretely, the spread configurations suggest that there are configurations in favor of each considered objective. Solutions with low spread fail to propose multiple alternative trade-offs. However, as already stated in Section 6.5.4 diverse solutions with low quality are not meaningful in practice.

6.7.2 Threats to validity

Several threats to the validity of the present study are identified. Our results are based on five SPLs. Hence, it is possible that our conclusions do not generalize to other cases. To reduce this threat, we took four different and large FMs with different number of features and constraints. In particular, the FMs we chose have a varied density of constraints, i.e., the number of constraints per feature vary. We used both slightly and heavily constrained models such as Linux and uClinux with an average of respectively 50 and 1.3 constraints per feature. Another threat is due to the randomness involved in the approaches studied. Indeed, there is a chance that the observed results happened by chance. To reduce this threat, we performed each approach 30 times independently, thereby reducing the influence of random effects. Another threat is identified due to potential errors, unknown parameters or differences in the implementation. In addition, the machines used may influence the results. To reduce this threat, we performed a careful verification of our results and several manual tests at all stages of our implementation. Additionally, we make publicly available both our implementation and our data. Finally, a threat is due to the artificial way the values of the attributes were assigned, i.e., the actual usage scenario, and to the replication of the state-of-the-art. Unfortunately, there is

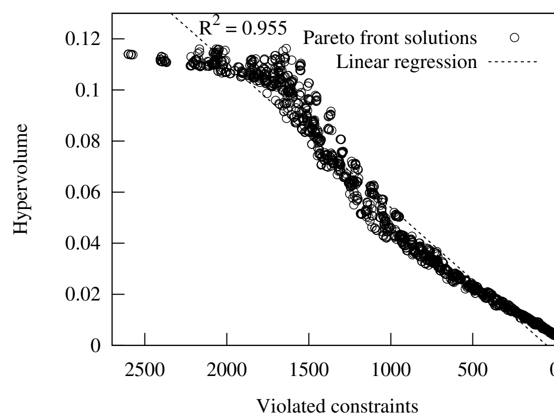


Figure 6.2: Impact of the violated constraints on the hypervolume for Linux. As solutions tend to conform to the constraints of the model, the optimization of the other objectives degrades.

no available implementation of the previous work [SIMA13b]. To overcome this issue, we carefully replicated and verified, multiple times, all parameters and the technical details of the experiments as described in [SIMA13b]. Additionally, we used the same framework, algorithms and settings as the previous work.

6.8 Conclusions

In this chapter, we presented a multi-objective configuration approach handling testing objectives targeting a single configuration. We have demonstrated that our SPL optimization approach, SATIBEA, significantly outperforms the current state-of-the-art with maximal effect size. We also provide results that show that it is important to include constraint solving techniques in SPL optimization approaches and that our technique scales to the largest SPLs hitherto considered in the literature. Since reproducibility has been identified as a central tenet of the research in software engineering [WRH+00], we make the source code of our implementation and our experimental data publicly available at `http://research.henard.net/SPL/ICSE_2015/`.

7

Handling multiple testing objectives targeting a whole configuration suite

In the previous chapter, we presented an approach for generating configurations according to multiple testing objectives. However, these objectives were targeting a single configuration, .e.g., maximizing the number of optionnal features of a configuration. In this chapter, we propose a generalized approach for generating a configuration suite according to objectives related to multiple configurations, such as the cost or the number of these configurations.

This chapter is based on the work that has been published in the following paper:

- Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Multi-objective test generation for software product lines. In *Proceedings of the 17th International Software Product Line Conference*, pages 62–71. ACM, 2013

Contents

7.1 Introduction **88**
7.2 The multi-objective configuration generation approach **88**
7.2.1 Modeling individuals and population 88
7.2.2 Modeling the genetic algorithm operations 89
7.2.3 The objective function 89
7.2.4 Overview of the approach 90
7.3 Case study **92**
7.3.1 Approach parameters 92
7.3.2 Evaluation of the objective function **F** (research question 1) 93
7.3.3 Comparison with Random (research question 2) 95
7.4 Threats to validity **97**
7.5 Conclusions **98**

7.1 Introduction

As we have seen in the previous chapter, finding a SPL configuration satisfying multiple and potentially conflicting objectives is a difficult problem. The problem is even more difficult if we aim at generating a configuration suite fulfilling testing objectives which cover a set of configurations, which is the problem that we aim at tackling.

Contributions of this chapter. In this chapter, we introduce a multi-objective genetic algorithm specially adapted for SPLs. Our approach combines genetic algorithms and constraint solving techniques in a complementary way to generate configuration suites to test that simultaneously fulfill three testing objectives : 2-wise coverage, testing costs and number of configurations. In brief, this chapter brings the following outcomes:

- We model the configuration generation problem for SPLs as a search problem. The proposed approach models configurations as genes and sets of configuration as individuals. It also suggests some possible operations on the individuals and an objective function. Therefore, it enables search-based approaches to solve the configuration generation problem.
- We use constraint solving technique to prune the invalid configurations from the search space. This is a crucial step towards enabling an efficient search process.
- We propose a genetic algorithm to solve the multi-objective optimization problem. The conducted study show that the approach is practically effective and feasible.

The remainder of this chapter is organized as follows. Section 7.2 details the introduced algorithm to solve multi-objective configuration generation for SPLs. Section 7.3 reports on experiments and Section 7.4 discusses the threats to validity. Finally, Section 7.5 concludes the chapter.

7.2 The multi-objective configuration generation approach

The proposed approach is a multi-objective genetic algorithm. Like any genetic algorithm, it requires the definition of its ingredients (genes, individuals and population), its operations (selection, crossover and mutation) and the objective function that evaluates how each individual fits to the problem.

7.2.1 Modeling individuals and population

A solution to our problem is a configuration suite that gives the maximum 2-wise coverage with the minimum cost and number of configurations. To fit the problem with the genetic algorithm, it is needed to model the population, the individuals and the genes in terms of the actual problem. Therefore, since an individual I represents a possible solution to the problem, it can be modeled as a configuration suite $I = \{C_1, ..., C_m\}$. Thus, each valid configuration represents a gene and the set of individuals handled by the genetic algorithm represents the population. This allows forming as a search space all the possible sets of valid configurations. This represents a huge space due to the intractable number of the possible configurations contained in a SPL.

However, enabling a search approach over this space cannot be performed directly. Recall that not all the configurations of a SPL form valid ones. Therefore, there is a need to efficiently deal with the

invalid configurations. This is not an easy task due to the large number of invalid configurations, especially for large SPLs [POS+12]. To overcome this difficulty, a SAT solver is used to provide random valid configurations. This is achieved by randomizing the solutions' enumeration order of the FM's formula [LBP10]. To this end, random configurations, sets of random configurations and the initial population can be produced efficiently [HPP+12]. Thus, the search space is reduced to only include valid configurations. The importance of this step is that it prunes the invalid configurations from the search space.

7.2.2 Modeling the genetic algorithm operations

Crossover is an operation defined between two selected individuals, called the parents and it is performed as depicted by Figure 7.1. This operation is performed by selecting l configurations from the smallest in size parent and swapping them with randomly selected ones from the other (bigger in size) parent. Our individuals form sets of configurations and thus the order of the genes does not matter. Hence, swapping randomly some configurations is equivalent to the usual crossover operation. Additionally, doing so ensures that the individuals are having the same sizes during the whole evolution process. Crossover operation results in two offsprings. These are then mutated according to the mutation operation as depicted by Figure 7.2. In *mutation* operation, a configuration is randomly selected from the individual and replaced by a randomly selected configuration (from the space of all the valid configurations).

Besides the above operations, the proposed approach incorporates two additional operations. These are the elitism and diversify operations. *Elitism* selects the best e individuals of one population and includes them directly to the new one. *Diversify* operation adds one new individual, randomly produced directly into the new population. This ensures the diversity of the population individuals during the evolution process.

7.2.3 The objective function

The proposed approach is based on an objective function $\mathbf{F}(x)$, specially designed for the SPL testing context. As introduced in Chapter 2, Section 2.3.3, $\mathbf{F}$ is a vector composed of the following $k = 3$ objective functions F_1, F_2 and F_3.

1. Maximization of the 2-wise coverage. This objective aims at ensuring that the selected configurations have the highest possible level of 2-wise coverage:

$$F_1(x) = cov(x),$$

   where cov is a function that evaluates the number of pair of features covered by $x = \{C_1, ..., C_m\}$.

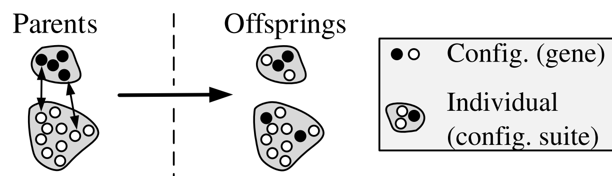


Figure 7.1: Crossover operation. A random number of l configurations are selected in the smallest parent. Each of them is swapped with a random configuration selected from the other parent to produce the two offsprings.

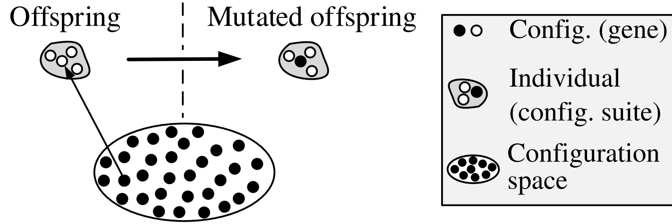


Figure 7.2: Mutation operation. A random configuration is selected from the offspring and replaced by a configuration randomly selected from the configuration space.

2. Minimization of the number of configurations. Here, the objective is to test the minimum number of configurations:

$$F_2(x) = card(x),$$

where *card* is a function that returns the number of configurations m of $x = \{C_1, ..., C_m\}$.

3. Minimization of the testing cost. This objective function aims at minimizing the cost of testing the configurations:

$$F_3(x) = cost(x),$$

where *cost* is a function returning the cost of testing these $x = \{C_1, ..., C_m\}$ configurations.

In order to evaluate **F**, each objective function is normalized so that they have the same magnitude using the following formula [MA04]:

$$\frac{F_i(x) - F_i^*}{F_i^{\max} - F_i^*},$$

where F_i^* is the utopia point and $F_i^{\max}$ the maximum objective functions values. In addition, the objective max F_1 is transformed into a minimization problem $\min(-F_1)$ in order to deal with minimization problems only. As a result, each objective function returns a value that holds between 0 and 1, where 0 means that the objective is perfectly fulfilled. To evaluate **F**, each function is assigned a weight w_j, where $\sum_{j=1}^{k} wj = 1$. Thus, the fitness of each individual $I = \{C_1, ..., C_m\}$ is computed using a weighted sum as follows:

$$\mathbf{F}(I) = \sum_{j=1}^{k} w_j F_j(I).$$

7.2.4 Overview of the approach

The technique is formalized in Algorithm 5. Informally, this approach starts by creating an initial random population (lines 3 to 14). The size of the population is specified by the user as long as the maximum size of an individual. Each individual is a set of $1, ..., m$ configurations randomly selected from the space of all the configurations that are valid towards the FM (lines 6 to 11). The objective function is then evaluated for each individual of the initial population (line 12).

The second step of the algorithm is the evolution of the population into a new one (lines 15 to 46). First, the elitism operation is performed (lines 17 to 19). Then, one random individual is created, evaluated and added to the new population to ensure having new configurations (lines 20 to 27). This is the diversity operation. The next steps ares crossover and mutation. To complete the new population until reaching its size n, individuals are created using crossover and mutation operators. The crossover (lines 29 and 30) aims at creating two offsprings from their selected parents. The two

Algorithm 5 Multi-objective configuration suite generation

```
input: t, n, m, e < n, ℂ_mutation, w_1, .., w_k, fm    ▷ t is the time or number of iterations, n is the population size, m
  is the maximum individual size, e is the number of individuals involved in elitism, ℂ_mutation is the probability to
  mutate one individual, w_1, .., w_k are the respective weights of each objective function F_1, .., F_k and fm is the FM.
output: x = {C_1, ..., C_m}                                         ▷ Solution (an Individual)
x ← ∅
pop ← ∅                                                             ▷ Population is a set of individuals
while card(pop) < n do
    s ← random integer from 1, ..., m
    I ← ∅                                                           ▷ An individual is a set of s configurations
    while card(I) < s do
        P ← random configuration(fm)                                ▷ Using a SAT solver
        I ← I ∪ {P}
    end while
    Evaluate F(I) = Σ_{j=1}^{k} w_j F_j(I)
    pop ← pop ∪ {I}
end while
while elapsed time or number of iterations < t do
    newPop ← ∅
    while card(newPop) < e do
        newPop ← newPop ∪ {{I} | I ∈ pop ∧ I ∉ newPop ∧ min F(I)}
    end while
    s ← random integer from 1, ..., m
    I ← ∅
    while card(I) < s do
        P ← random configuration(fm)                                ▷ Using a SAT solver
        I ← I ∪ {P}
    end while
    Evaluate F(I) = Σ_{j=1}^{k} w_j F_j(I)
    newPop ← newPop ∪ {I}
    while card(newPop) < n do
        I_parent1, I_parent2 ← selection(pop)      ▷ Selected according to a fitness proportionate selection method
        I_child1, I_child2 ← crossover(I_parent1, I_parent2)
        C_1, C_2 ← random real number from [1, 2]
        if C_1 ≤ ℂ_mutation then
            mutate(I_child1)
        end if
        if C_2 ≤ ℂ_mutation then
            mutate(I_child2)
        end if
        Evaluate F(I_child1) = Σ_{j=1}^{k} w_j F_j(I_child1)
        Evaluate F(I_child2) = Σ_{j=1}^{k} w_j F_j(I_child2)
        newPop ← newPop ∪ {I_child1)} ∪ {I_child2)}
    end while
    while card(newPop) > n do
        newPop ← newPop \ {{I} | I ∈ newPop ∧ max F(I)}
    end while
    pop ← newPop
end while
x ← I | I ∈ pop ∧ min F(I)
return x
```

parents are selected using a fitness proportionate selection, also known as roulette wheel selection. The mutation occurs on the offsprings with a certain probability fixed by the user (lines 32 to 37). The fitness of these two offsprings is then evaluated and these two new individuals are added to the new population (lines 39 to 40).

Finally, the new population is reduced to the initial population size (lines 42 to 44) and the current population is replaced by the new one (line 45) and it continues to the next generation (line 46). When the algorithm terminates, the individual that has the best fitness, i.e. the one with the minimum **F** value is returned (line 47).

Table 7.1: Feature models used in the case study.

	Counter Strike Simple FM	DS Sample	SPL SimulES, PnP	Electronic Drum	Smart Home v2.2	Video Player	Model Transformation	Coche Ecologico
#Features	24	32	41	52	60	71	88	94
#Configurations	18,176	73,728	6,912	331,776	3.87×10^9	4.5×10^{13}	1.65×10^{13}	2.32×10^7
2-sets	833	1,448	2,592	3,746	6,189	7,528	13,139	11,075

7.3 Case study

In this section, the proposed multi-objective configuration generation approach is assessed on a set of FMs. The objective of this case study is to answer the two following research questions:

- [RQ1] *Is* **F** *capable of leading to a fulfillment of the three objectives? In other words, does the minimization of* **F** *results in a maximization of* F_1 *(or a mininimization of* $(-F_1)$*), a minimization of* F_2 *and a minimization of* F_3*?*
- [RQ2] *How does the multi-objective generation technique compare with a random one?*

Answering the first question amounts to evaluate whether the objective function **F** is capable of improving the studied objectives. We expect to see a decreasing trend in all three objectives in relation to population generations. In practice, this means that a better trade-off can be achieved. This trade-off leads to a higher 2-wise coverage, less configurations and a lower cost. Since no other approach takes into account these objectives at the same time, our second question aims at comparing the two of them when keeping the other one set, to enable the comparison with random test suite generation. Hence, we select random configuration sets of a) the same size and b) achieving the same 2-wise coverage as our approach. If, for the same number of configurations, our approach achieves to provide a lower cost and higher 2-wise coverage than the random set, we can consider it as being a better one. Similarly, it will be successful if it provides less configurations and a lower cost than random for a certain level of 2-wise coverage.

To answer these questions, an experiment composed of 8 FMs of varying sizes was conducted. We applied our approach on these FMs to evaluate the population evolution and to compare it with a random approach. All the employed FMs were taken from the Software Product Line Online Tools (SPLOT) repository [MBC09] and have been widely used in literature. The FMs details are recorded in Table 7.1. For each subject FM, the number of features, the number of configurations that can be configured and the number of valid pairs are presented. For each FM, we randomly assigned a value between 1 and 10 to all non-mandatory features to represent the cost value of the features, as presented in Chapter 2, Section 2.1.3.3. Further details on the conducted experiment are given in the following subsections.

7.3.1 Approach parameters

Since objective F_1 results in selecting a higher number of configurations and F_2, F_3 results in selecting a lower number of configurations, we assigned the following weights: $w_1 = 0.5$ for F_1 and $w_2 = w_3 = 0.25$ for F_2 and F_3. This assignment represents the balanced between the studied objectives as set for

Table 7.2: Evolution of the objective function **F** and the sub-objectives from the initial generation of the multi-objective approach to the final one (500 generations). The final and initial values are the average between the 30 runs. The *p*-value is the results of the Mann-Whitney U Test between the 30 first initial values and the 30 final ones.

	F			F_1: 2-wise cov. (to maximize)			F_2: # config. (to minimize)			F_3: cost (to minimize)		
	Initial	Final	*p*-value	Initial	Final	*p*-value	Initial	Final	*p*-value	Initial	Final	*p*-value
C. Strike FM	0.163	0.115	<0.001	819.7	819.63	0.79	15.46	14.466	0.176	658.13	369.90	<0.001
SPL SimulES	0.163	0.136	<0.001	1431.4	1439.03	0.003	14.33	12.8	<0.001	906.73	680.66	<0.001
DS Sample	0.189	0.172	<0.001	2364.2	2382.9	0.07	31.866	27.7	<0.001	1040.2	887.96	<0.001
Elec. Drum	0.146	0.132	<0.001	3633.6	3665.06	<0.001	18.7	17.4	0.04	1221.96	1079.6	0.001
Smart Home	0.177	0.138	<0.001	6041.46	6056.66	0.60	17.7	17.03	0.33	2282.86	1537.46	<0.001
Video Player	0.162	0.135	<0.001	7430.66	7428.76	0.20	15.13	13.86	0.011	2000.63	1443.86	<0.001
Model Trans.	0.175	0.153	<0.001	12733.73	12788.1	0.387	17.96	17.16	0.48	3522.5	2829.36	<0.001
Coche Eco.	0.169	0.154	<0.001	10560.26	10618.06	0.039	21.13	19.66	0.19	2083.1	1761.63	<0.001

our experiment. It is noted that our approach is not limited to this balance. Thus, the tester may set a different balance according to his needs. The population size has been set to $n = 100$ and the maximum size of an individual (a potential solution) has been set to $m = 100$. The mutation probability $\mathbb{P}_{\text{mutation}}$ has been set up to 0.05 and the elitism value e to 5. Finally, the approach has been limited to run for $t = 500$ generations.

7.3.2 Evaluation of the objective function **F** (research question 1)

7.3.2.1 Setup

We performed the multi-objective configuration generation 30 times per FM using the above-mentioned parameters. For each of the 30 runs, we measured the initial values (at generation 1) and the final values (at generation 500) of both the 3 sub-objectives and the objective **F**.

To evaluate whether these differences are statistically significant, we followed the guidelines suggested by Arcuri and Briand in [AB11] by performing a Mann-Whitney U Test. It is a non-parametric statistical hypothesis test for assessing whether one of two samples of independent observations tends to have larger values than the other. We obtain from this test a probability called *p*-value which represents the probability that the two samples are equal. It is conventional in statistics to consider that the difference is not significant if the *p*-value is higher than the 5% level. The experiments involving this statistical test used two-tailed tests.

7.3.2.2 Results

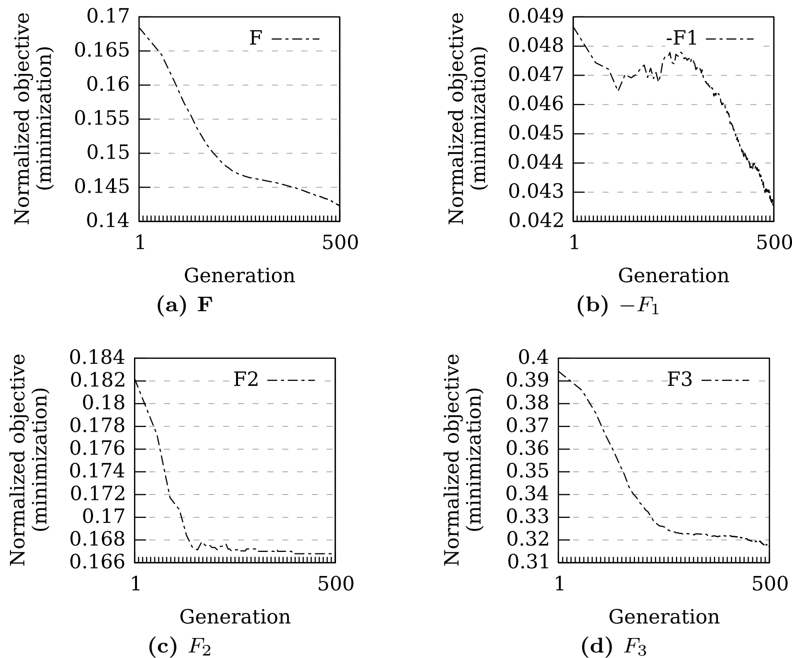
Table 7.2 presents per FM the average values on the 30 runs for each of the objective and for **F**. F_1 is the number of pairs covered by the generated configurations (to maximize), F_2 is the number of configurations (to minimize) and F_3 is the cost of testing the generated configurations (to minimize). **F** is the compromised between the 3 objectives. From this table, one may observe that final values of both the 3 objectives and the objective function are better than initial one, i.e. decreasing for F_2, F_3 and **F** and increasing for F_1 since it's a maximization. This underlines that a decreasing in **F** leads to a better fulfillment of each objective. These difference are most of the time statistically significant with *p*-values lower than 0.05 or highly significant with *p*-values lower than 0.001, fact

which demonstrates the appropriateness of the objective function with only 500 generations of the algorithm.

Besides, Figure 7.3 depicts the evolution of the objective function **F** and the normalized objective function over the generations of the algorithm. Since all the three objectives are transformed into minimization problems, i.e. lower values of the objective functions represent better solutions to the problem, this figure clearly shows the decreasing trend of each objective function. It therefore demonstrates that **F** leads to a better solution regarding all the examined objectives.

7.3.2.3 Answering research question 1

The results presented in the previous section clearly show the ability of the objective function to fulfill the three studied objectives. In particular, **F** is capable of finding better solutions for all the objectives under investigation. While some differences may not be statistically significant, recall that the approach is a compromise between the conflicting objectives. It therefore tends to compromise the 3 objectives according the w_1, w_2 and w_3 parameters. The overall objective **F** has always highly statistically significance difference, showing that **F** clearly guides the population generation. Finally, it must be mentioned that the proposed approach achieves the above results using only a small number of generations (500 generations). This can be viewed as an achievement of the approach since search-based approaches do require thousands of executions in order to be effective [HM10].



(a) **F** **(b)** $-F_1$ **(c)** F_2 **(d)** F_3

Figure 7.3: Evolution of the objective function **F** and each normalized sub-objectives (to be minimized) during the 500 generations of the multi-objective approach.

7.3.3 Comparison with Random (research question 2)

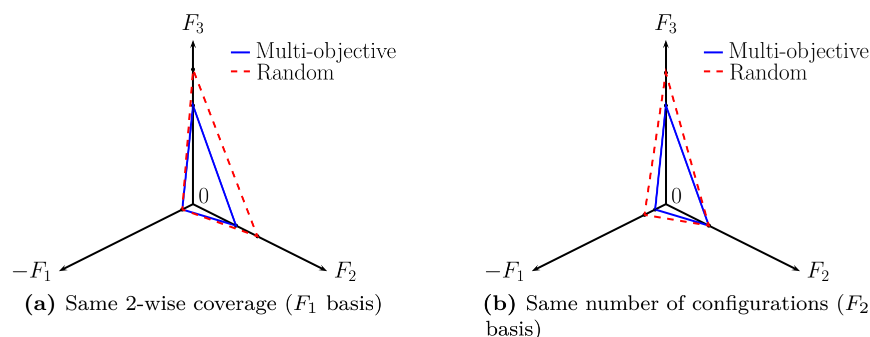
7.3.3.1 Setup

To assess our approach, we compared it with a baseline. To do so, we used two baseline comparison basis. In the first one, we selected random sets of configurations having the same F_1 value as our approach. In the second one we selected random configuration sets having the same F_2 value as our approach. The F_1 comparison basis aims at evaluating how many configurations for which cost are provided by the examined approaches (baseline and proposed) to achieve the same level of 2-wise coverage. The F_2 comparison basis evaluates the 2-wise coverage and the cost induced by the generated configurations for the same number of configurations. In the end, for each run of our approach, two random runs have been performed: the first one by setting F_1 as the comparison basis and the second one using F_2. The conducted experiment (including both the baseline and the proposed approach) was independently repeated 30 times.

To evaluate whether the differences are significant, we performed a Mann-Whitney U Test, as presented in Section 7.3.2.1. For each comparison ($\mathbf{F}, F_2$ and F_3 on F_1 comparison basis and $\mathbf{F}, F_1$ and F_3 on F_2 comparison basis), we got one p-value per FM, i.e. 8 in the total. Each p-value results from the comparison between the 30 values obtained on the 30 runs by the proposed approach with those obtained at random.

7.3.3.2 Results

Table 7.3 records the comparison between our configuration generation approach and the baseline one based on the F_1 and the F_2 comparison basis. For each FM and each comparison basis, the average, minimum and maximum values of $\mathbf{F}$ and the three objectives F_1, F_2 and F_3 one the 30 runs are presented. F_1 represents the number of pairs covered by the generated configurations (to be maximized), F_2 represents the number of configurations (to be minimized) and F_3 represents the cost of testing the generated configurations (to be minimized). From this table, it is clear that the proposed approach performs better than a random one. For instance, for the Smart Home v2.2 FM on the F_1 comparison basis (i.e. for achieving the same 2-wise coverage), the proposed approach



(a) Same 2-wise coverage (F_1 basis)

(b) Same number of configurations (F_2 basis)

Figure 7.4: Normalized sub-objectives (to be minimized) according to F_1 and F_2 comparison basis. Values closer to 0 represent better solutions. These values are the average on all the feature models for all the 30 runs of each approach. The length of each axis is 1.

Table 7.3: Comparison between the multi-objective approach and the random one. For each feature model, the values of the objective functions studied and **F** are represented. F_1 is the number of pairs (to be maximized), F_2 is the number of configurations (to be minimized) and F_3 is the cost (to be minimized). The comparison with random has been made by fixing either F_1 or F_2 on 30 runs per approach.

			Multi-objective approach			Random		
			avg	min	max	avg	min	max
C. Strike FM	Same 2-wise coverage (F_1 basis)	**F** (to min.)	0.115	0.107	0.122	0.173	0.154	0.2
		F_2 (to min.)	14.466	11	17	18.833	12	28
		F_3 (to min.)	369.9	276	440	832.433	512	1,218
	Same # of configurations (F_2 basis)	**F** (to min.)	0.115	0.107	0.122	0.181	0.155	0.238
		F_1 (to max.)	819.633	803	828	806.833	763	828
		F_3 (to min.)	369.9	276	440	665.13	510	889
SPL SimulES	Same 2-wise coverage (F_1 basis)	**F** (to min.)	0.136	0.13	0.14	0.173	0.159	0.187
		F_2 (to min.)	12.8	10	15	18.966	14	26
		F_3 (to min.)	680.666	567	813	1,234.8	940	1,656
	Same # of configurations (F_2 basis)	**F** (to min.)	0.136	0.13	0.14	0.177	0.16	0.20
		F_1 (to max.)	1,439.033	1,429	1,446	1,411.066	1,367	1,445
		F_3 (to min.)	680.666	567	813	859.333	680	1,039
DS Sample	Same 2-wise coverage (F_1 basis)	**F** (to min.)	0.172	0.169	0.177	0.214	0.182	0.302
		F_2 (to min.)	27.7	22	34	44.966	32	83
		F_3 (to min.)	887.966	700	1,106	1,469.8	1,024	2,716
	Same # of configurations (F_2 basis)	**F** (to min.)	0.172	0.169	0.177	0.214	0.195	0.246
		F_1 (to max.)	2,382.9	2,328	2,428	2,236.366	2,093	2,362
		F_3 (to min.)	887.966	700	1,106	902.9	725	1,121
Elect. Drum	Same 2-wise coverage (F_1 basis)	**F** (to min.)	0.132	0.130	0.133	0.155	0.142	0.173
		F_2 (to min.)	17,4	14	20	24.5	17	32
		F_3 (to min.)	1,079.6	872	1,255	1,645.533	1.165	2,210
	Same # of configurations (F_2 basis)	**F** (to min.)	0.132	0.130	0.133	0.155	0.144	0.180
		F_1 (to max.)	3,665	3,628	3,693	3,585.133	3,458	3,661
		F_3 (to min.)	1,079.6	872	1,255	1,174.7	926	1,367
Smart Home	Same 2-wise coverage (F_1 basis)	**F** (to min.)	0.138	0.133	0.144	0.191	0.166	0.234
		F_2 (to min.)	17.033	12	20	21.966	15	36
		F_3 (to min.)	1,537.466	1,195	1,836	3,016.533	1,974	5,184
	Same # of configurations (F_2 basis)	**F** (to min.)	0.138	0.133	0.144	0.19	0.166	0.223
		F_1 (to max.)	6,056.666	5,973	6,107	5,976	5,756	6,087
		F_3 (to min.)	1,537.466	1,195	1,836	2,330.4	1,532	2,872
Video Player	Same 2-wise coverage (F_1 basis)	**F** (to min.)	0.135	0.128	0.138	0.167	0.152	0.188
		F_2 (to min.)	13.866	11	16	16.5	14	24
		F_3 (to min.)	1,443.866	1,230	1,687	2,2236.5	1,858	3,339
	Same # of configurations (F_2 basis)	**F** (to min.)	0.135	0.128	0.138	0.173	0.159	0.208
		F_1 (to max.)	7,428.766	7,739	7,468	7,341.233	6,925	7.471
		F_3 (to min.)	1,443.866	1,230	1,687	1907.433	1,467	2,444
Model Trans.	Same 2-wise coverage (F_1 basis)	**F** (to min.)	0.153	0.149	0.158	0.185	0.176	0.199
		F_2 (to min.)	17.166	14	21	20,733	18	25
		F_3 (to min.)	2,829.366	2,353	3,319	4,325.166	3,588	5,304
	Same # of configurations (F_2 basis)	**F** (to min.)	0.153	0.149	0.158	0.187	0.174	0.199
		F_1 (to max.)	12,788.1	12,657	12,902	12,595.3	12,262	12,815
		F_3 (to min.)	2,829.366	2,353	3,319	3,556.166	2,742	4,509
Coche Eco.	Same 2-wise coverage (F_1 basis)	**F** (to min.)	0.154	0.151	1.158	0.186	0.172	0.217
		F_2 (to min.)	19.666	16	24	29.3	20	41
		F_3 (to min.)	1,761.633	1383	2153	2,984,3	2,051	4,384
	Same # of configurations (F_2 basis)	**F** (to min.)	0.154	0.151	1.158	0.188	0.167	0.207
		F_1 (to max.)	10,618	10,492	10,726	10,302	10,040	10,553
		F_3 (to min.)	1,761.633	1383	2153	2,016.333	1,631	2,404

proposes on average around 17 configurations with a cost of ≈ 1,537 where a random technique requires around 22 configurations with a cost of ≈ 3,016.

Figure 7.4 depicts the achieved values for each objective for both basis of comparison. The values are the average on all the FMs for the 30 runs. Here, the smallest triangle signifies a better solution since each normalized objective is a function to be minimized. An objective value equals to 0 means that this objective is perfectly fulfilled. The length of the axis of each objective is 1. This figure shows that a) for the same 2-wise coverage, the proposed approach requires less configurations with a lower cost and b) for the same number of configurations, our approach provides a higher 2-wise coverage and a lower cost compared to random configurations.

Finally, the results of the statistical test are depicted by Figure 7.5. It presents, for each comparison, the distribution of the 8 p-values (one per FM). Each p-value is the result of the comparison between the 30 values obtained for each objective during each run of the proposed and random approaches. From this figure, one may observe that all the p-values are lower to 0.001, fact which denotes the a high statistical difference between the results achieved by our approach compared to the results of the random one.

7.3.3.3 Answering research question 2

We compared the multi-objective configuration generation approach with a baseline technique using F_1 and F_2 as a basis comparison. In all the cases, the objectives are better fulfilled by our approach, as demonstrated by the results of Section 7.3.3.2. Overall, for the same 2-wise coverage, the approach selects less configurations with a lower cost. For the same number of configurations, the approach provides a higher 2-wise coverage at a lower cost. In addition, the differences between the objectives values reach by our technique and the values reached by the baseline are statistically highly significant, fact which demonstrates the effectiveness of the approach.

7.4 Threats to validity

The conducted experiments involve potential threats to validity. First, there is a threat regarding the generalization of the results reported in this study. Indeed, a different set of FMs might output

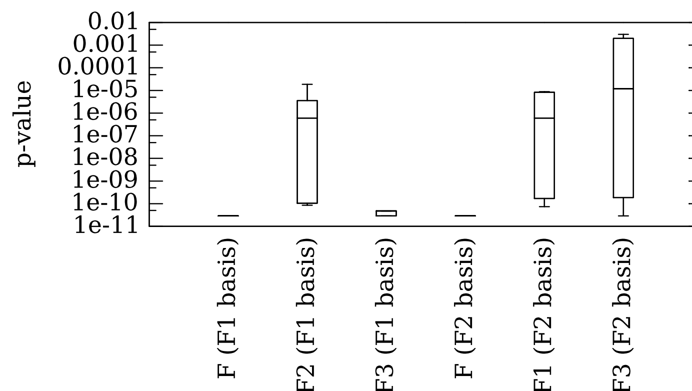


Figure 7.5: Distribution of the p-values for the comparison with random. For each comparison ($\mathbf{F}, F_2$ and F_3 on F_1 comparison basis and $\mathbf{F}, F_1$ and F_3 on F_2 comparison basis), the 8 p-values (one per feature model) are represented with a boxplot. Each p-value has been obtained by comparing the 30 values obtained on the 30 runs for each approach.

different results. We used a set of 8 FMs widely used in literature with different size and level of complexity to reduce this threat and to ensure that the FMs used form a good sample.

Additional threats can be identified due to a) to our implementation, which might contains errors that can affect the presented results and b) the performed experiments. To overcome this issue, we divided our implementation into subroutines to minimize the potential errors and we make it publicly available. We also repeated the conducted experiments independently for 30 times to avoid any risk due to random effects, like the fortunate selection of the (nearest) optimal solution.

7.5 Conclusions

Optimizing different objectives is a hard problem due to the presence of conflicts between them. For example, minimizing the number of tests is in conflict with the maximization of their 2-wise coverage since generally more tests lead to a higher coverage. This chapter tackled this problem by proposinga generalized search-based approach handling multiple testing objectives. Finally, to enable the reproducibility of our results, we make the source code of our approach and the data used for the experiments publicly available at `http://research.henard.net/SPL/SPLC_2013/`.

Part IV

Configuration evaluation

8

Assessing configurations with mutation and application to similarity testing

The previous parts aimed at generating configurations. The following chapters introduce way to evaluate them prior testing. In this chapter, a mutation analysis approach performed on feature model, which allows evaluating the quality of configurations, is presented. The similarity heuristic presented in Chapter 4 is also evaluated.

This chapter is based on the work that has been published in the following paper:

- Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Assessing software product line testing via model-based mutation: An application to similarity testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 188–197. IEEE, 2013

Contents

8.1 Introduction **102**
8.2 Approach **103**
8.2.1 The mutation analysis approach for evaluating configurations 103
8.2.2 configuration suite generation 103
8.2.3 Evaluation of the quality of the configuration suite 104
8.3 Experiments **104**
8.3.1 Evaluation of the mutation score depending on the type of configurations . . . 104
8.3.2 Impact of the similarity-based prioritization on the mutation score 106
8.3.3 Answering research questions 1 and 2 108
8.3.4 Threats to validity 109
8.4 Conclusions **109**

8.1 Introduction

The underlying idea of this chapter is to use the information provided by FMs to evaluate the quality of configurations. In SPL context, mutants can be used to either produce configurations, as performed in Chapter 5, or to evaluate them, which is the objective of this chapter. This leads to our first RQ:

[RQ1] *How can mutation analysis be performed on model-based SPLs in order to evaluate the configurations quality?*

The use of mutation in literature is twofold. First, it has been used to generate tests [DO91, PM12]. Second, it has been used to evaluate other testing approaches [ABLN06, ABL05]. We focus on the second part. In our context, a configuration suite represents a set of configurations and a mutant can be considered as a fault. In model-based testing, it has been found that dissimilar configuration suites have a higher fault detection power than similar ones [HB10]. This similarity heuristic can be used to reduce the size of the configuration suites by removing similar configurations, as shown in the Chapter 4. This approach is particularly useful since for SPL, the number of configurations to test is usually enormous, with potentially billions of possible configurations to test [McG10]. Moreover, the benefit of this heuristic has not been thoroughly assessed in the context of SPL testing. It leads to our second RQ:

[RQ2] *Do dissimilar configuration suites have a higher mutant detection rate in the context of SPL and FM testing?*

To answer RQ1, we introduce a mutation analysis for SPLs based on FMs. Thus, we produce different erroneous variants of the original FM by introducing possible defects. Then, we evaluate configuration suites generated from the original FM towards the modified FMs. To answer RQ2, we use a similarity heuristic [HPP+12] to compare two configurations and to evaluate the similarity degree of a given configuration suite. An experiment conducted on both similar and dissimilar configuration suites towards FMs of different size demonstrate the higher ability of dissimilar configuration suites to detect the defect embodied in the modified FMs. Further, the validity of a similarity-driven prioritization technique [HPP+12] is also evaluated.

Contributions of this chapter. In brief, the present chapter provides the following contributions:

- A mutation analysis approach for SPLs based on FMs,
- An experimentation performed on real FMs from small to large scale ones, which (a) confirm the hypothesis that dissimilar configuration suites have a higher mutant detection rate than similar ones and (b) assess a similarity-driven prioritization technique.

The remainder of this chapter is organized as follows: Section 8.2 details the mutation testing and similarity approaches. Section 8.3 reports on the conducted experiments. Finally, Section 8.4 concludes the chapter.

8.2 Approach

8.2.1 The mutation analysis approach for evaluating configurations

In this chapter, we introduce a mutation testing approach for SPLs based on FMs. The approach works as follows. From a FM represented as a Boolean formula, we produce several erroneous versions of this model by applying mutant operators on the clauses of the formula. These erroneous versions of the original FM are the mutants. Then, using a SAT solver [LBP10], we generate configurations from the original FM and we check their validity towards the mutants. This evaluation is performed by checking whether the generated configurations satisfy or not the Boolean formula of the mutants. This process allows evaluating the quality of the configuration suite through the computation of the MS. The approach is depicted by Figure 8.1.

We propose two mutation operators which perform at the clause level of the Boolean formula of the FM. These two operators are summarized in Table 8.1. The first operator takes a clause c_i and randomly change a literal of this clause into its negation. As a result, this operator alters an existing clause of the FM formula. The second operator aims at creating two clauses from a given one by replacing one of the disjunction operator in this clause by a conjunction operator. Thus, this second operator creates two clauses from an existing one, increasing the total number of clauses of the Boolean formula by one.

8.2.2 configuration suite generation

We use a SAT solver [LBP10] to generate configurations randomly from the configuration space using the unpredictable approach described in Chapter 4. configurations randomly generated were found to be dissimilar due to the large size of the search space. Besides, to generate similar configurations, one configuration is randomly selected from the search space of all the valid configurations. Then, adjacent configurations to the randomly selected one are retrieved. These are configurations sharing many selected or unselected features in common.

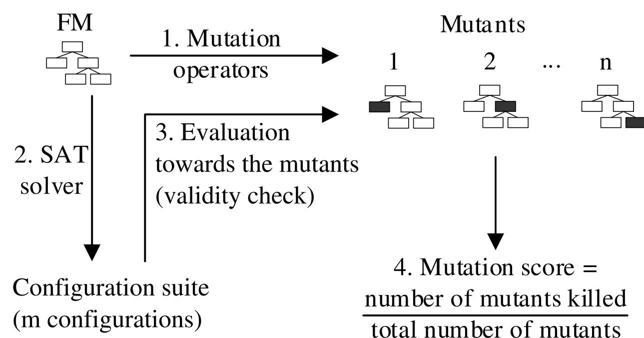


Figure 8.1: Mutation analysis approach for evaluating configurations.

Table 8.1: Mutation operators for feature models. The first operator negates one of the variable of the constraint. The second operator splits the constraints into two by removing one of the disjunction operators.

Input (FM clause)	**Applies on**	**Result**
$c_i = f_1 \vee ... \vee f_k \vee ... \vee f_m$	a literal: $f_k, k \in [1, m]$	$c'_i = f_1 \vee ... \vee \neg f_k \vee ... \vee f_m$
$c_i = f_1 \vee ... \vee f_k \vee ... \vee f_m$	a disjunction operator	$c'_i = f_1 \vee ... \vee f_k$ and $c''_i = f_{k+1} \vee ... \vee f_m$

Table 8.2: 12 Various size feature models.

	C. Strike FM	DS Sample	Elec. Drum	Smart Home v2.2	Video Player	Model Transformation	Coche Ecologico	Printers	Electronic Shopping	eCos 3.0 i386pc	FreeBSD kernel 8.0.0	Linux kernel 2.6.28.6
#Features	24	41	52	60	71	88	94	172	290	1,244	1,396	6,888
# Config.	18,176	6,912	331,776	3.9E9	4.5E13	1.7E13	2.3E7	1.1E27	4.5E49	N/A	N/A	N/A

8.2.3 Evaluation of the quality of the configuration suite

Here, we try to link the MS of the examined configuration suites with the quality of the configuration suite in terms of dissimilarity between the configurations. To this end, w use the similarity heuristic and the prioritization technique respectively presented in Chapter 4, Section 4.2 and Algorithm 2.

8.3 Experiments

In this section, the mutation testing approach of FMs and the evaluation of the quality of the generated configuration suites are assessed. The experimental study employs 12 real FMs from two common repositories [MBC09, lin]. These FMs are recorded in Table 8.2. It presents, for each FM, the number of features it contains and the total number of configurations that can be configured from the model.

8.3.1 Evaluation of the mutation score depending on the type of configurations

The first experiment aims at evaluating the impact of the quality of the configuration suite on the MS. In other words, the objective is to evaluate whether dissimilar configuration suites kill more mutants than similar ones.

8.3.1.1 Setup

We generated 100 mutants for each of the 12 FMs used in this case study. The chance to produce a mutant with one of the two mutation operators was set to 0.5. We generated three type of

Table 8.3: Mutation score achieved with different types of configuration suites (%).

	#Config.	2		10			50		
	Type	Dissim.	Sim.	Dissim.	Half Sim./Dissim.	Sim.	Dissim.	Half Sim./Dissim.	Sim.
C. Strike	avg	69.50	53.02	82.29	80.09	64.40	84.83	84.44	76.86
	min	54	31	77	75	45	83	83	63
	max	77	71	85	84	75	85	85	84
DS Sample	avg	50.64	35.91	69.26	64.71	52.72	88.67	86	77.07
	min	46	22	60	56	39	81	76	66
	max	58	50	81	75	63	90	90	90
Elec. Drum	avg	63.59	45.92	81.31	78.99	60.04	83	83	80.05
	min	55	33	75	73	45	83	83	76
	max	69	58	83	83	69	83	83	83
Smart Home	avg	78.18	58.83	93.13	91.71	66.49	94	93.86	77.62
	min	62	34	89	85	46	94	93	63
	max	90	76	94	94	79	94	94	86
Vid. Player	avg	69.16	53.64	82.50	80.70	58.36	84.02	83.64	67.68
	min	31	25	77	57	29	83	77	34
	max	81	72	85	84	73	86	85	77
Model Tran.	avg	68.69	52.33	83.17	80.24	54.21	84	83.99	58.94
	min	60	39	80	66	41	84	83	45
	max	74	65	84	84	66	84	84	73
Coche Eco.	avg	72.66	59.21	83.93	80.40	60.90	88.94	88.32	66.46
	min	67	49	79	71	47	87	85	55
	max	78	68	88	86	69	89	89	74
Printers	avg	59.58	45.13	76.45	72.68	47.47	82.50	80.95	56.48
	min	38	21	72	60	31	80	77	38
	max	70	60	81	77	63	84	83	66
Elec. Shop.	avg	66.27	48.33	86.42	82.80	49.09	89.23	88.70	54.18
	min	55	38	80	76	38	88	85	40
	max	77	60	90	89	63	90	90	67
eCos	avg	60.35	48.32	77.83	73.22	48.41	83.49	81.13	49.64
	min	49	38	74	65	38	77	76	40
	max	68	65	86	83	56	87	87	57
FreeBSD	avg	26.82	18.62	40.91	36.55	18.89	46.89	45.28	19.14
	min	10	5	32	25	5	45	40	7
	max	37	32	46	43	29	48	48	32
Linux	avg	10.14	7.14	15.72	13.97	7.40	23.21	21.40	7.29
	min	7	3	11	10	4	14	14	4
	max	17	16	24	22	16	35	34	10

configuration suites: configuration suites containing only dissimilar configurations, configuration suites containing half similar and dissimilar configurations, and configuration suites containing only similar configurations. Different size of tests suites were generated for each of these types: configuration suites of 2, 10 and 50 configurations. We evaluated the configuration suites towards the 100 mutants to compute the MS. The generation of the configuration suites and the evaluation of the MS has been repeated 100 times.

8.3.1.2 Results

The results are recorded in Table 8.3. It presents, for each FM, the average, minimum and maximum MS achieved for the different size and types of configuration suites. Following this table, one may observe that the MS for the configuration suites of dissimilar configurations is higher than the tests

suites, containing both similar and dissimilar configurations, and the latter is higher than similar configuration suites. In some cases, like for the Linux kernel 2.6.28.6 FM with configuration suites of 50 configurations, the MS achieved by dissimilar configuration suites is more than three time bigger than the MS reached by similar configuration suites.

To evaluate whether these differences are statistically significant, we followed the guidelines suggested by Arcuri and Briand in [AB11] by performing a Mann-Whitney U Test. It is a non-parametric statistical hypothesis test for assessing whether one of two samples of independent observations tends to have larger values than the other. We obtain from this test a probability called *p*-value which represents the probability that the two samples are equal. It is conventional in statistics to consider that the difference is not significant if the *p*-value is higher than the 5% level.

For each size of configuration suites and for each of the 100 executions, we took the MS achieved by the dissimilar and similar configuration suites for each FM. We thus have on the one hand the 12 MSs for the similar configuration suites, and on the other hand the 12 MSs of the dissimilar configuration suites. It leads to 100 *p*-values corresponding to the number of executions performed. The results are presented in Figure 8.2. It represents via a box plot the distribution of the 100 *p*-values resulting of the Mann-Whitney U test between the MS achieved by similar and dissimilar configuration suites for the 100 executions. From this figure, it can be observed that the difference is statistically significant for configuration suites of 10 and 50 configurations since all the *p*-values are lower to the significance level of 5%.

8.3.2 Impact of the similarity-based prioritization on the mutation score

Here, the objective is to assess whether the similarity-driven prioritization [HPP^{+}12] is effective. In other words, we want to evaluate whether k configurations selected according to the prioritization technique presented in Section 8.2 kill more mutants than k configurations randomly prioritized.

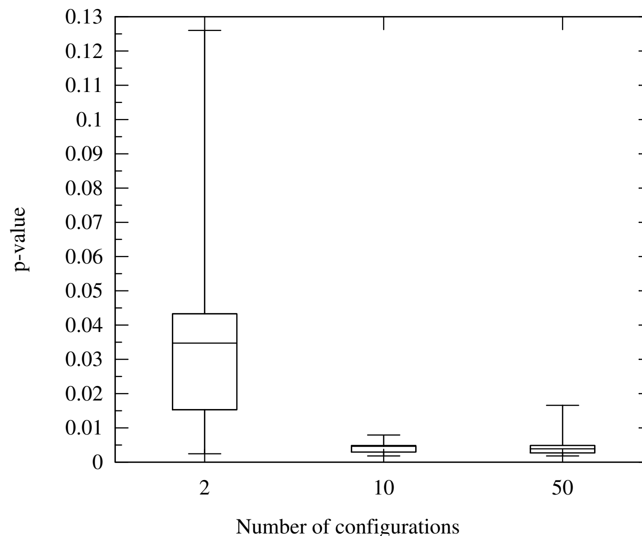


Figure 8.2: Distribution of the 100 *p*-values resulting of the Mann-Whitney U Test between the mutation score achieved by similar and dissimilar configuration suites for the 100 executions.

Table 8.4: Area under curve observed for the two prioritization techniques.

	Counter Strike Simple FM	DS Sample	Electronic Drum	Smart Home v2.2	Video Player	Model Transformation	Coche Ecologico	Printers	Electronic Shopping	eCos 3.0 i386pc	FreeBSD kernel 8.0.0	Linux kernel 2.6.28.6
Similarity	8,346	8,568	8,178	9,277	8,209	8,281	8,730	8,029	8,778	8,081	4,541	2,304.5
Random (min)	7,995	7,666	7,983	9,118	8,010	8,040	8,290	7,615	8,343	7,477	4,014	1,473.5
Random (avg)	8,187	8,119	331,776	9,175	8,143	8,183	8,514	7,802	8,535	7,773	4,240	1,883
Random (max)	8,314	8,473	331,776	9,269	8,205	8,275	8,685	8,022	8,730	8,068	4,531	2,272.5

8.3.2.1 Setup

For each FM, we generated tests suites of 100 configurations containing both similar and dissimilar configurations. We executed the similarity-driven prioritization technique to prioritize each configuration suite. Then, we applied 100 times a random prioritization of the configurations in order establish a random ordering of them. Finally, for each number of k configurations selected between 0 and 100, we evaluated the MS achieved with these k configurations. To compare the prioritization approaches, the area under curve is evaluated.

8.3.2.2 Results

Table 8.4 presents the area under curve for the similarity and random prioritizations for each FM. From this table, one can see that the similarity-driven technique bestow a higher area under curve value than the random one, fact which demonstrates its effectiveness. Indeed, in any cases, the similarity-driven prioritization achieves the highest area under curve value. Figure 8.3 depicts the curve of the MS achieved for different number of configurations selected, averaged on the 12 FMs. This figure also shows the benefit of the similarity-driven prioritization. For instance, a MS of around 80% can be achieved with the similarity-driven prioritization with only around 5 configurations while the random one requires around 20 configurations. In addition, only around 30 similarity prioritized configurations are needed to achieve the maximum score of around 85% where the random prioritization requires 100 configurations.

To evaluate whether the differences between the similarity-driven prioritization technique and the random one are significant, we performed a Mann-Whitney U Test. For each FM, we compared the results of the similarity prioritzation with each of the 100 random executions. It thus leads to 100 p-values per FM, which are represented with box plot in Figure 8.4. From this figure, one can see that the results are not significantly significant for the small FMs. One explanation is that only a small number of configurations, e.g. 5 or 10 allows killing most of the mutants, and thus the remaining configurations don't kill any new mutants, leading to two samples which are almost the same. However, for the largest FMs, the difference is significant, with median values greatly below the significance level of 5%.

8.3.3 Answering research questions 1 and 2

The mutation testing approach proposed in this chapter aims at evaluating the quality of the tests. We produced 100 erroneous FMs and we evaluated the fault detection power of different type of configuration suites generated from the correct FM. The approach uses mutation operators which

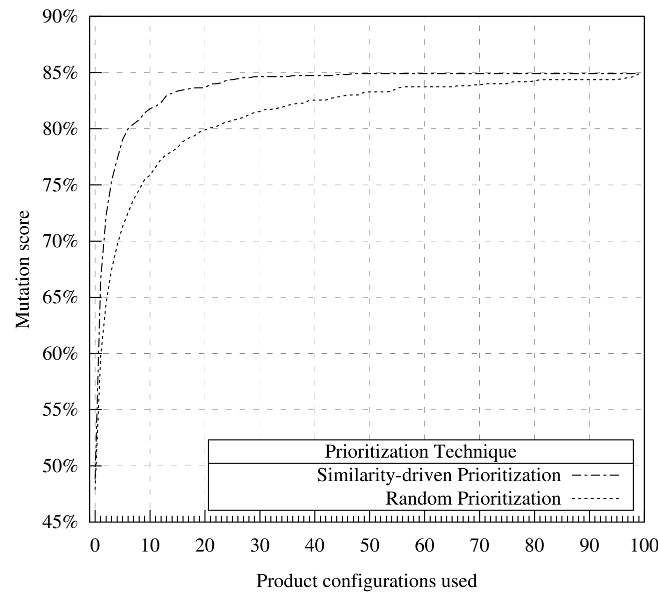


Figure 8.3: Mutation score achieved for the prioritization techniques averaged on the 12 feature models.

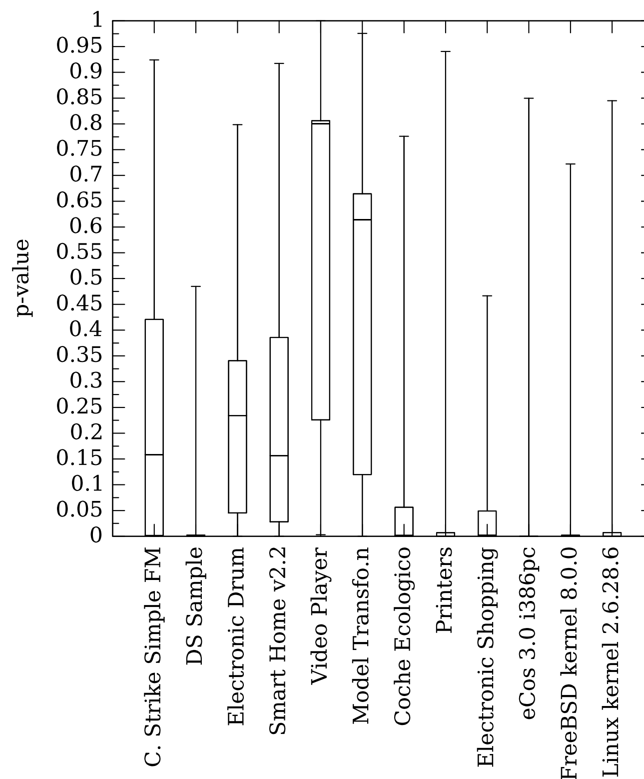


Figure 8.4: Distribution of the 100 p-values resulting of the Mann-Whitney U Test between the mutation score achieved by configurations prioritized with the similarity technique and the 100 random prioritization.

perform on the Boolean formula of the FM by altering clauses. The results obtained show that the tests are able to kill some mutants, which makes the approach interesting for testing FMs.

The impact of dissimilar and similar configuration suites on the MS is clear. The results obtained in this chapter show that dissimilar configuration suites bestow a higher mutant detection rate than similar ones. Indeed, both the evaluation of the MS depending on the type of tests and the similarity-driven prioritization showed that dissimilar configurations kill more mutants than similar ones. In addition, we observed a significant statistical difference between the MS achieved by the different type of configuration suites, fact which confirm the similarity hypothesis.

8.3.4 Threats to validity

First, there is an *external validity* threat. Indeed, we cannot ensure that the mutation analysis and prioritization approaches will output analogous results on different sets of FMs, e.g. larger or more constrained. To reduce this threat, we used 12 FMs of different sizes, from 24 to almost 7,000 features. Each of these FM bestow a different number and complexity regarding their constraints.

Besides, an *internal validity* threat could be due to potential errors in our implementation which could affect the presented results. To overcome these threats, we divided the implementation into sub stages. This practice allowed having a better control on each of the steps composing the proposed approaches. Besides, to avoid any risk due to random effects like coincidental selection of mutants or tests, we repeated the experiments 100 times.

Finally, whether the defects introduced in the mutants reflects real faults form a *construct validity* threat. Mutation has proven to be effective and the mutation operators used performs on the logical constraints of the FM. These constraints linking the features represent a potential source of errors in the model's construction stage.

8.4 Conclusions

In this chapter, we presented a mutation analysis approach for SPLs based on FMs. To the best of our knowledge, it is the first mutation analysis approach applied in the context of SPLs. In addition, this approach has been evaluated towards similar and dissimilar configuration suites to evaluate whether dissimilar configuration suites bestow a higher mutant detection rate than similar ones. The benefit of dissimilar configuration suite is that they allow to drastically decrease the number of configurations to test.

Our experiments, performed on 12 real FMs of different size demonstrate the effectiveness of the approach. In particular, the higher ability of dissimilar configuration suites to kill mutants has been proven with both the MS and prioritization evaluations. Indeed, dissimilar configuration suites are in some case able to kill two or three times more mutants than similar configurations. The prioritization results emphasized the benefit of this heuristic, showing that testing first dissimilar configurations rather than similar ones allow killing more mutants.

9

Mutation analysis as a potential alternative to combinatorial interaction testing

The previous chapter introduced an approach for assessing configuration suites with model-based mutation. This chapter evaluates whether the mutation criterion used in Chapter 5 and Chapter 8 can form a viable alternative to combinatorial testing by evaluation its correlation with fault detection.

This chapter is based on the work that has been published in the two following papers:

- Mike Papadakis, Christopher Henard, and Yves Le Traon. Sampling program inputs with mutation analysis: Going beyond combinatorial interaction testing. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, pages 1–10, Washington, DC, USA, 2014. IEEE Computer Society
- Christopher Henard, Mike Papadakis, and Yves Le Traon. Mutalog: A tool for mutating logic formulas. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '14, pages 399–404, Washington, DC, USA, 2014. IEEE Computer Society

Contents

9.1 Introduction . . . 112
9.2 Example . . . 113
9.2.1 Case 1: absence of input constraints . . . 113
9.2.2 Case 2: presence of input constraints . . . 115
9.3 Test suite evaluation . . . 115
9.3.1 The program input model . . . 115
9.3.2 The combinatorial interaction testing approach . . . 116
9.3.3 The mutation approach . . . 116
9.4 Experimental methodology . . . 117
9.4.1 Definition of the experiment . . . 117
9.4.2 Subjects . . . 118
9.4.3 Evaluating test suites . . . 118
9.4.4 Rank correlation analysis . . . 119
9.5 Experimental results . . . 120
9.5.1 Correlation analysis (research questions 1 & 2) . . . 120
9.5.2 Comparing combinatorial testing and mutation (research question 3) . . . 120
9.6 Discussion . . . 122
9.6.1 Additional consideration about mutation and combinatorial testing . . . 123
9.6.2 Cost of the approach . . . 123
9.6.3 Threats to validity . . . 124
9.7 Conclusions . . . 124

9.1 Introduction

Mutation as an alternative to CIT. The present chapter applies mutation analysis to the CIT input model. Thus, it introduces an alternative but more representative measure of the effectiveness of a test suite. Traditionally, mutation analysis is applied at the program code and aims at evaluating the quality of a test suite [Off11]. It operates by introducing artificial defects, called *mutants*, in the code of the tested program. Thus, multiple versions of the program under test are produced. Each version contains a defect that is introduced by making a slight modification. The test suites are then evaluated based on their ability to distinguish the introduced problems [Off11]. Contrary to the traditional approach, we apply mutation on the input model. Hence, we do not need to execute the system. We only need to evaluate whether the selected test cases satisfy or violate the altered input models. The testing process can then be performed based on the selected test cases.

In this chapter, we introduce defects on the model of the program inputs. Thus, we create various input models, each one containing one defect. We apply this approach in the same way as t-wise testing is applied. However, instead of measuring the number of covered interactions, we measure the number of mutated input models that are violated by the selected tests. Therefore, we have test cases that satisfy all the constraints of the original input model but which violate the constraints of the mutated ones. The number of the mutated models having constraints violated by the test cases to the total number of the mutated models represents the effectiveness ability of these tests.

The question that it is investigated here is whether mutation analysis can provide a good indication about the quality of the test suites. A positive answer to this question will indicate that the proposed approach is valid and will motivate practitioners to use it. However, as already mentioned, CIT forms the mainstream approach to select and evaluate such test cases. Thus, another question that need to be answered is whether mutation analysis provides better estimations than the CIT about the fault detection ability of the test suites. Therefore, the main contribution of the present chapter is the comparison of the proposed mutation approach with the CIT one according to their ability to expose faults. Currently, only a few works investigate the fault detection ability of the interaction testing in the presence of input constraints and none focusing on mutating the input models.

Contributions of this chapter. We present results of a controlled experiment that involves four real world programs with input constraints. The utilized programs are widely used in experimental studies and are accompanied by a set of faulty versions. Therefore, we can evaluate the ability of the examined approaches to predict the actual fault detection of the selected test suites. This is performed based on a rank correlation analysis. The findings of the study reveal that both the mutation and CIT approaches are good predictors of actual fault detection. This is in line with the previous research on CIT. However, it turns out the the mutation-based approach generally provides better estimations than CIT. This difference is significant in most of the cases, thus indicating a strong correlation between code-level faults and the proposed model-level defects.

In brief, the contributions of this chapter are the following:

- We propose a mutation analysis approach applied at the program input level to assess the quality of test suites,
- We evaluate the correlation between a) the number of interactions covered and b) the number of the introduced mutants distinguished by a test suite with its actual fault detection. We find out that the model-based defects have a stronger correlation with code-level faults than the input parameter interactions.

The remainder of this chapter is organized as follows: Section 9.2 demonstrates the application of the CIT and the proposed approach through an example and Section 9.3 details them. Sections 9.4

and 9.5 details the experiment and present its findings respectively. Section 9.6 discusses some issues regarding the mutation approach. Finally, Section 9.7 concludes the chapter.

9.2 Example

This section introduces an illustrative example to explain how mutation analysis can be applied. The approach is based on a model of the program inputs, as those typically used by CIT approaches, e.g., [PYCH13]. Such a model encompasses the different parameters and the constraints between these parameters.

Consider the following model M involving three parameters p_1, p_2 and p_3. Each parameter is a variable of the model. The parameter p_1 can take the two values a and b, p_2 can take the three values c, d and e and the last parameter p_3 can only take the f value. Thus, the model M is defined as follows:

$$M := p_1 \in \{a, b\},\ p_2 \in \{c, d, e\} \text{ and } p_3 \in \{f\}.$$

Typically, input models involve input constraints between the parameters. For sake of simplicity, we will first present the approach in the case where there are no input constraints, case 1). Then, the general case that involves input constraints, case 2), will be demonstrated.

9.2.1 Case 1: absence of input constraints

9.2.1.1 Flattening the model

In order to apply mutation analysis, we flatten this model to a Boolean one denoted as M_b. The flattened model involves 6 variables instead of three, which corresponds to the values of the parameters: a_{p_1}, b_{p_1}, c_{p_2}, d_{p_2}, e_{p_2} and f_{p_3}. Each of this variable is Boolean, i.e., it can take only two values, $true$ or $false$. We then need the to add to M_b constraints which specify that only one value can be selected at a time for a given parameter. For instance, a_{p_1} and b_{p_1} cannot be both $true$ because it would mean that $p_1 = a$ and $p_1 = b$ at the same time. Following our example, we need to add to M_b 8 following constraints:

$$(a_{p_1} \Rightarrow \neg b_{p_1}),\ (b_{p_1} \Rightarrow \neg a_{p_1}),\ (c_{p_2} \Rightarrow \neg d_{p_2}),\ (c_{p_2} \Rightarrow \neg e_{p_2}),\ (d_{p_2} \Rightarrow \neg c_{p_2}),\ (d_{p_2} \Rightarrow \neg e_{p_2}),\ (e_{p_2} \Rightarrow \neg c_{p_2}),\ (e_{p_2} \Rightarrow \neg d_{p_2}).$$

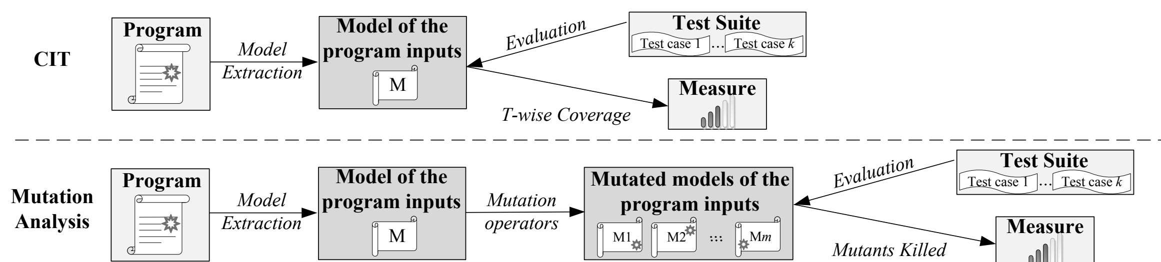


Figure 9.1: The mutation analysis approach compared to the traditional combinatorial interaction testing.

Thus, the Boolean model M_b equivalent to M is defined as:

$M_b := a_{p_1} \in \{true, false\}$, $b_{p_1} \in \{true, false\}$, $c_{p_2} \in \{true, false\}$, $d_{p_2} \in \{true, false\}$, $e_{p_2} \in \{true, false\}$, $f_{p_3} \in \{true, false\}$, $(a_{p_1} \Rightarrow \neg b_{p_1})$, $(b_{p_1} \Rightarrow \neg a_{p_1})$, $(c_{p_2} \Rightarrow \neg d_{p_2})$, $(c_{p_2} \Rightarrow \neg e_{p_2})$, $(d_{p_2} \Rightarrow \neg c_{p_2})$, $(d_{p_2} \Rightarrow \neg e_{p_2})$, $(e_{p_2} \Rightarrow \neg c_{p_2})$, $(e_{p_2} \Rightarrow \neg d_{p_2})$.

9.2.1.2 Creating mutants of the flattened model

The next step consist in creating defective (i.e., mutated) versions of the M_b model. To produce such a mutated model, we alter one of the constraint of M_b. It creates a different version of this model where the defect is the change operated on the constraint. This process is repeated several times to create various mutated versions of M_b. For instance, the constraint $C = (a_{p_1} \Rightarrow c_{p_2})$ of M_b can be altered to $C' = (a_{p_1} \Rightarrow \neg c_{p_2})$ while producing a new mutated model from M_b.

In the following, we consider the two following mutants, in which the altered constraint of M_b is underlined:

- $M'_b := a_{p_1} \in \{true, false\}$, $b_{p_1} \in \{true, false\}$, $c_{p_2} \in \{true, false\}$, $d_{p_2} \in \{true, false\}$, $e_{p_2} \in \{true, false\}$, $f_{p_3} \in \{true, false\}$, $(a_{p_1} \Rightarrow \neg b_{p_1})$, $(b_{p_1} \Rightarrow \neg a_{p_1})$, $(c_{p_2} \Rightarrow \neg d_{p_2})$, $(c_{p_2} \Rightarrow \neg e_{p_2})$, $(d_{p_2} \Rightarrow \neg c_{p_2})$, $(d_{p_2} \Rightarrow \neg e_{p_2})$, $\underline{(e_{p_2} \Rightarrow c_{p_2})}$, $(e_{p_2} \Rightarrow \neg d_{p_2})$.
- $M''_b := a_{p_1} \in \{true, false\}$, $b_{p_1} \in \{true, false\}$, $c_{p_2} \in \{true, false\}$, $d_{p_2} \in \{true, false\}$, $e_{p_2} \in \{true, false\}$, $f_{p_3} \in \{true, false\}$, $(a_{p_1} \Rightarrow \neg b_{p_1})$, $(b_{p_1} \Rightarrow \neg a_{p_1})$, $(c_{p_2} \Rightarrow \neg d_{p_2})$, $(c_{p_2} \Rightarrow \neg e_{p_2})$, $(d_{p_2} \Rightarrow \neg c_{p_2})$, $(d_{p_2} \Rightarrow \neg e_{p_2})$, $(e_{p_2} \Rightarrow \neg c_{p_2})$, $\underline{(e_{p_2} \Rightarrow f_{p_3})}$.

9.2.1.3 Evaluating program inputs

The proposed approach selects program inputs which satisfy the constraints of M_b and at the same time do not satisfy the constraints of the mutated models. For instance, consider the two following program inputs:

- $I_1 = \{a_{p_1} = true, b_{p_1} = false, c_{p_2} = true, d_{p_2} = false, e_{p_2} = false, f_{p_3} = true\}$,
- $I_2 = \{a_{p_1} = true, b_{p_1} = false, c_{p_2} = false, d_{p_2} = false, e_{p_2} = true, f_{p_3} = true\}$.

We simplify the representation of these inputs to consider only the values selected, i.e., equals to *true*:

- $I_1 = \{a_{p_1}, c_{p_2}, f_{p_3}\}$,
- $I_2 = \{a_{p_1}, e_{p_2}, f_{p_3}\}$.

Both I_1 and I_2 satisfy M_b. We evaluate each input towards each mutant. I_1 satisfies the two mutants, I_2 satisfies the second mutant M''_b but violates M'_b. Indeed, the underlined constraint of M'_b is violated: selecting $e_{p_2} = true$ implies selecting $c_{p_2} = true$, but I_2 has $e_{p_2} = true$ and $c_{p_2} = false$. We thus identify that the I_2 test case is effective in finding the introduced defect. Measuring the number of such defects found by a test suite serves as an effectiveness measure to our approach. We can say that I_1 did not violated any of the two mutants and that I_2 violated half of the mutants. Thus, with respect to our approach, I_2 is more effective.

With respect to CIT, for instance 2-wise interactions, I_1 covers 15 interactions. An example of such an interaction is $(a_{p_1} = true, b_{p_1} = false)$. The total number of 2-wise interactions of the model M_b is 66. Thus, the CIT measure for I_1 is $\frac{15}{66}$. Consider now the two following test suites:

- $T_1 = \{I_2\}$,

- $T_2 = \{I_1, I_2\}$.

With respect to our approach, both the test suites are similarly effective since they both violates one of the two mutants. Thus, we measure $\frac{1}{2}$ for both T_1 and T_2, which is the number of violated mutants to the total ones.

With respect to CIT, T_1 covers $\frac{15}{66}$ 2-wise interactions while T_2 covers $\frac{24}{66}$. As a result, for CIT, the second test suite is more effective as it covers more interactions than the first one.

9.2.2 Case 2: presence of input constraints

When there are input constraints in the model M, we simply transform them to Boolean ones and we add them to the constraints of M_b. For instance, suppose M contains the input constraint $((p_1 = a) \Rightarrow (p_2 = c))$. It is transformed into $(a_{p_1} \Rightarrow c_{p_2})$. Thus, in this case, the Boolean model of M is:

$M_b := a_{p_1} \in \{true, false\}$, $b_{p_1} \in \{true, false\}$, $c_{p_2} \in \{true, false\}$, $d_{p_2} \in \{true, false\}$, $e_{p_2} \in \{true, false\}$, $f_{p3} \in \{true, false\}$, $(a_{p_1} \Rightarrow \neg b_{p_1})$, $(b_{p_1} \Rightarrow \neg a_{p_1})$, $(c_{p2} \Rightarrow \neg d_{p2})$, $(c_{p2} \Rightarrow \neg e_{p2})$, $(d_{p2} \Rightarrow \neg c_{p2})$, $(d_{p2} \Rightarrow \neg e_{p2})$, $(e_{p2} \Rightarrow \neg c_{p2})$, $(e_{p2} \Rightarrow \neg d_{p2})$, $\underline{(a_{p_1} \Rightarrow c_{p2})}$.

The added input constraint is underlined. The process then continues similarly as the case 1), by mutating M_b (see Section 9.2.1.2).

9.3 Test suite evaluation

The global process of the proposed mutation approach is depicted by Figure 9.1. The technique operates on a model of the program inputs, presented in Section 9.3.1, by creating defective (i.e., mutated) model versions. The application of the approach, detailed in Section 9.3.3,is equivalent to CIT since it uses the same input models. However, instead of measuring the interactions covered by the test cases, as done by CIT (presented in Section 9.3.2), it measures the ability of the test cases to distinguish the defective versions.

9.3.1 The program input model

Applying CIT or the proposed approach requires building a model of the program inputs. This model represents the different test cases that can be derived by combining the program inputs. Thus, the model encompasses the different parameters of the program, their values, and the constraints that link them. It can be seen as a set of constraint, where each constraints involves variables (the parameters), their possible values and how they can be combined.

A constraint regulates the use of the input parameters. For instance, a constraint may denote that setting a specific parameter p_i with a specific value v prevents another parameter p_j from taking the value w. We can formalize this constraint as $(p_i = v) \Rightarrow (p_j \neq w)$. Thus, a program input composed of both $p_i = v$ and $p_j = w$ does not *satisfy the constraint* since p_j should not be set to w when p_i is set to v_i. We say that a test case *satisfies the model of the program inputs* if all the constraints of this model are satisfied at the same time.

Each constraint involves several variables that correspond to the parameters of the program. A variable has a domain which corresponds to the values that the parameter can take. For instance, the variable p_i may take two values v and w. In that case, p_i has a domain involving two values, v and w.

In practice, our approach requires a flattening of the model to make it Boolean. This is a typical process undertaken by most of the CIT tools, e.g., [CCL03]. It also gives the opportunity to the mutation approach to produce mutants in the case that no input constraints exists. Instead of having a model with variables corresponding to parameters, the flattened model contain variables that represent all the possible values for all the parameters. Each one of these variables can be *true* or *false*, depending on whether this variable is assigned an input value. For instance, if a parameter can take three different values, the flattening transforms the parameter variable, which has a domain of three values into three different variables. Doing so transforms the model to a Boolean one as required by the proposed approach.

9.3.2 The combinatorial interaction testing approach

The CIT approach works by counting the unique number of interactions (or combinations) between any t parameters values exercised by the test suite. Such interactions are called t-wise interactions, as they involve combinations between the t parameters. Thus, given the input model, we evaluate all the possible t combinations of the input parameters. Then, based on the input constraints, we eliminate the invalid ones, i.e., the combinations that are prohibited by the constraints. Finally, the effectiveness measure of the test suite is calculated based on the t-wise coverage which is the number of t combinations that are covered by the test cases.

9.3.3 The mutation approach

The mutation approach operates by altering the Boolean model of the program inputs. It actually produces defective models by altering the model constraints. Thus, it creates several versions of the model, called *mutants*. Each mutant contains only one altered constraint. The constraints are altered based on a set of syntactic rules called *mutation operators*. Thus, by applying the mutation operators on all the model constraints, we end up with the sought set of mutants.

The constraints of the flattened model are Boolean and they are represented as a disjunction between variables. Thus, each constraint C between k variables has the general form $C = \bigvee_{i=1}^{k} v_k$, where v_k is a variable (corresponding to one parameter's value) either set to *true* or *false*.

We employ the two mutation operators presented in Table 8.1 of Chapter 8. The first operator alters a constraint by taking one of its variables and negating it. In other words, if the selected variable is

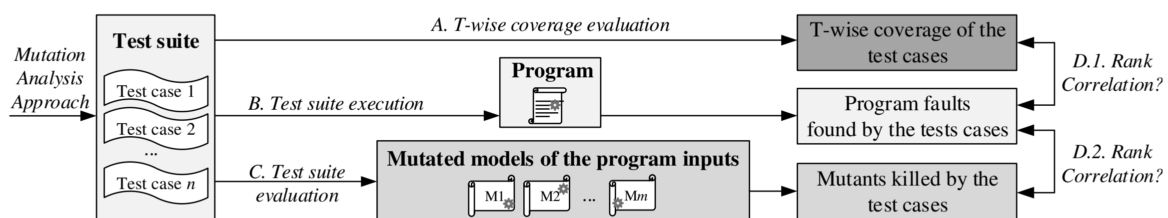


Figure 9.2: Experimental methodology for comparing the mutation analysis approach with combinatorial interaction testing.

true, it becomes *false*, and conversely. The second operator aims at creating two constraints from the original one. To this end, one of the disjunction operator in the constraint is replaced by a conjunction operator. Thus, this second operator splits a constraint into two, increasing by one the total number of constraints of the model.

A test suite is evaluated based on its ability to distinguish the defective models from the original one. We refer to the mutants that are distinguished by the test cases as *killed* and to those that can not be distinguished as *live*. However, how can we check whether a mutant is killed or not? To answer this question, we need to consider that our models are composed of Boolean constraints. Therefore, the evaluation is straightforward. It is examined whether a test case satisfies the constraints of the mutant model with a satisfiability (SAT) solver. The mutant is killed when its constraints are violated by the test case. In the opposite case it is live. It is noted that only the constraints of the mutant models can be violated. The constraints of the original model must always be satisfied in order to have valid program inputs. By determining the number of mutants that are killed by all the test cases, the overall effectiveness measure of the test suite is calculated.

9.4 Experimental methodology

The aims of the conducted experiment are summarized in the following RQs:

- [RQ1] *How well does the mutation-based approach evaluate the quality of the selected test suite?*
- [RQ2] *How well does the CIT approach evaluate the quality of the selected test suite?*
- [RQ3] *How does the mutation approach compares with CIT?*

Answering the first RQ is important in order to substantiate the practical use of mutation. Answering the second question is important in reinforcing the empirical evidence in favor of CIT.

Now, suppose that mutation and CIT are capable of predicting accurately the fault detection ability of test suites. In this case, practitioners will be able to evaluate the quality of their test suites. Going a step further, they will be able to prioritize their tests (by pointing first the test cases that cover the majority of the interactions or kill most of the mutants), reduce the suites size (by removing redundant tests) and guide the test generation process (by generating test cases that cover new interactions or kill additional mutants). However, in this case, which one should be used? This is our third RQ which aims at identifying which of the two examined approaches should be used in practice.

9.4.1 Definition of the experiment

To answer the stated RQs, we analyze the ability of test cases to cover t-wise interactions, to kill our mutants, and to expose code-based faults. We employ four subjects that are accompanied with test cases and faulty versions (Section 9.4.2). We then sample at random 30 test suites from the initial test suite of each program. Thus, we sample suites of random size from $4 \leq n \leq N$, where N is the size of the initial test suite. The minimum size of 4 test cases per suite has been chosen in order to have a sufficient sample for the correlation analysis. Then, we measure three metrics. a) the number of interactions covered, b) the number of mutants killed and c) the number of faults found (Section 9.4.3). These measures are recorded for every test case of the selected suites.

Then, these metrics are examined in order to identify possible correlations between them and to answer to RQs 1 and 2, Section 9.4.4. To this end, we perform a statistical analysis to quantify these

correlations. In other words, we try to measure the extent to which the relationship of covering interactions and killing mutants relates with fault detection. Thus, for each subject program, 30 *Kendall rank coefficients* are obtained by evaluating the correlation between the killed mutants and the faults found by the test cases, case denoted as MF, and 30 coefficients are obtained from the correlation between the t-wise coverage and the faults found. This latter comparison is denoted as tWF. In this work, we consider t-wise coverage for $t = 2, ..., 4$.

Finally, we compare the two methods based on the level of correlations, thus, answering to RQ3. An overview of the followed process is depicted by Figure 9.2.

9.4.2 Subjects

We use the four following programs: `flex`, `gzip`, `make` and `sed`. These subjects are taken from the Software-artefact Infrastructure Repository (SIR) [DER05] and their details are presented in Table 9.1. The examined versions of these program were randomly selected. Hence, for each subject, Table 9.1 records its size in uncommented lines of code [i], the number of faults per version taken from the faults matrix provided by the SIR, the number of variables and constraints of its respective model, the number of mutants and killable (i.e., there is at least one test configuration that cannot satisfy the faulty model) mutants obtained by applying the mutation operators, the number of the test cases contained in the initial test suite and the number of t-wise interactions for $t = 1, ..., 4$.

The input models are taken from the study of Petke et al. [PYCH13]. This study concerns the test case prioritization according to CIT and thus, their models are well suited for the present study. These models were built based on the descriptions of the Test Suite Specification Language (TSL) that are proposed with the utilized programs [PYCH13]. Thus, the parameters and values of these models represent the program input space. As described in Section 9.3, we transform these models to Boolean ones in order to evaluate the various test cases.

9.4.3 Evaluating test suites

9.4.3.1 T-wise

Given a test suite, we evaluate its t-wise coverage based on the following process. All the t-wise interactions between the parameters values covered by the first test case of the suite are recorded. Then, we consider the second test case and we add all the interactions that are not exercised by the first one. This process is repeated for all the test cases of the suite and it gives the cumulative number of unique t-wise interactions covered by each one of suite' test cases. Thus, given a test suite of n test cases $tc_1, ..., tc_n$, we obtain the t-wise coverage represented by the tuples (tc_i, c_i).

9.4.3.2 Fault detection

Given a test suite, we evaluate its fault detection ability based on the following process. We first take the faults found by the first test case of the suite by using the fault matrix provided by the SIR. This matrix contains the faults found by each test case. These faults are not considered while

[i] Measured with `cloc`: `http://cloc.sourceforge.net/`.

Table 9.1: The four subjects programs used in the experiments.

Subject		`flex`	`gzip`	`make`	`sed`
	v1	9,581	4,604	14,459	-
	v2	-	5,092	-	-
Uncommented lines of code	v3	-	-	-	7,161
	v4	11,470	-	-	-
	v7	-	-	-	14,177
	v1	19	16	19	-
	v2	-	7	-	-
Faults	v3	-	-	-	6
	v4	16	-	-	-
	v7	-	-	-	4
Model variables		23	29	20	34
Model constraints		43	91	21	143
Model mutants		139	295	63	527
Killable model mutants		139	292	63	373
Test cases		500	159	768	144
All 2-wise interactions		1,035	1,653	780	2,278
All 3-wise interactions		15,180	30,856	9,880	50,116
All 4-wise interactions		163,185	424,270	91,390	814,385
Valid 2-wise interactions		939	1,388	736	1,421
Valid 3-wise interactions		11,478	19,980	8,268	23,075
Valid 4-wise interactions		95,176	194,974	63,475	265,698

evaluating the next test cases. Then, the number of faults found by the second test case is recorded. This process is repeated for all the test cases of the test suite and provides the cumulative number of unique faults found by each test case. Thus, given a test suite of n test cases $tc_1, ..., tc_n$, we obtain the number of faults found after executing each test case. It is represented by the tuples (tc_i, f_i).

9.4.3.3 Mutation

We evaluate a test suite according to mutation based on the following process. Initially, the number of mutants killed by the first test case is determined. These mutants are removed and the second test case is evaluated according to all the remaining mutants. This process is repeated for the whole suite. Thus, the process gives the cumulative number of the unique killed mutants after executing each test case. Hence, given a test suite of n test cases $tc_1, ..., tc_n$, we obtain the number of mutants killed after executing each test case. It is represented by the tuples (tc_i, m_i).

9.4.4 Rank correlation analysis

Evaluating a test suite according to t-wise, fault detection and mutation, as described in the previous section gives the following information after executing $k \leq n$ tests of the test suite:

1. The current t-wise coverage achieved after considering the ith test case of the test suite,

2. The current number of faults found after executing the ith test case of the test suite,
3. The current number of mutated models that cannot be satisfied after considering the ith test case of the test suite.

Given these three measures, we evaluate whether 1) correlates with 2), whether 3) correlates with 2), and which of these two correlations is better. In order to evaluate these correlations, we compute the *Kendall τ rank correlation coefficient*. This coefficient is considered as the most robust and usefully interpreted statistical measure for this question [GGZ+13, Ken38, Cli96]. Thus, the coefficient is one one hand calculated given the correlation between the tuples (tc_i, c_i), (tc_i, f_i) and on the other hand given the tuples (tc_i, m_i) and (tc_i, f_i).

The Kendall τ is in the range $-1 \leq \tau \leq 1$. A coefficient of 1 indicates that the correlation between the two considered ranking is perfect. A coefficient around 0 denotes that the two observed sample are independent. Finally, a τ equals to -1 represents the complete absence of correlation between the two considered rankings.

9.5 Experimental results

This section reports results regarding the ability of the CIT and the mutation approaches to reveal actual faults. Then, it compares the two approaches.

9.5.1 Correlation analysis (research questions 1 & 2)

Figure 9.3 presents the distribution the Kendall coefficients for all the subject programs. MF denotes the coefficients resulting from the correlation analysis between the mutants killed and the faults found by the test cases. The correlation coefficient between the t-wise coverage of the test cases and the faults found on the program are denoted as tWF, with $t = 2, ..., 4$.

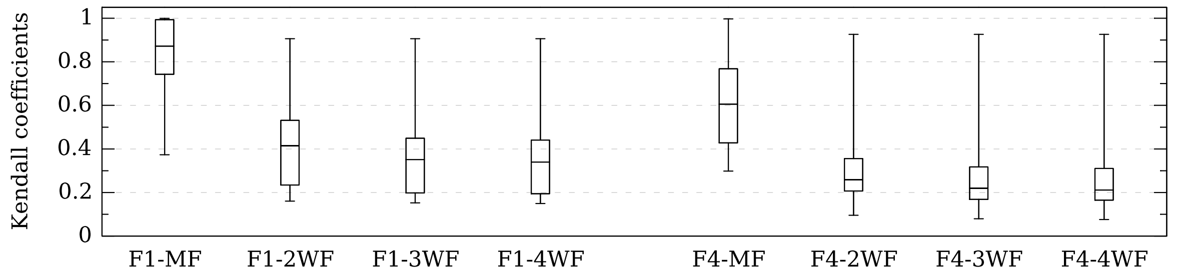
From these results we can infer three interesting conclusions. First, we observe that both CIT and mutation strongly correlate with fault detection for all the employed subjects. This is due to the fact that most of the coefficients are greater than 0.5 (median values) and almost all are at least 0.3, thus indicating very good correlations. Second, we observe big differences on the correlations between the subject programs and between the different versions. For example, `sed` v3 has coefficients close to 0.5 while `gzip` v2 is close to 0.2. Similarly, `gzip` has differences between v1 and v2 where the coefficients are close to 0.5 and 0.2 respectively. Third, we observe that higher t-values results in weaker correlations than lower t values in all the examined cases. The next section compares t-wise and CIT based on these results.

9.5.2 Comparing combinatorial testing and mutation (research question 3)

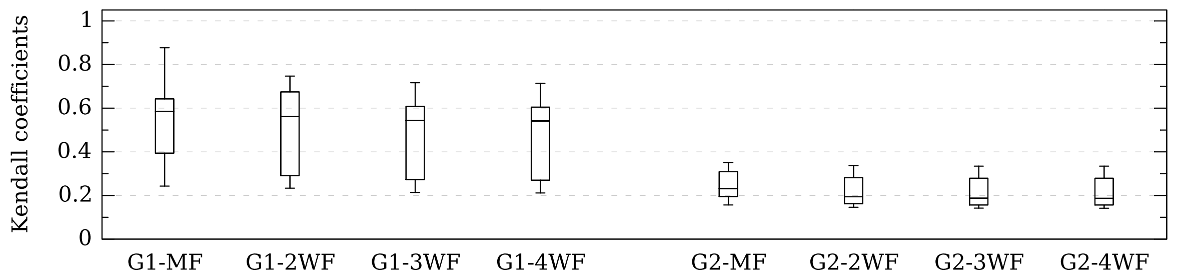
In order to compare the CIT approach with the mutation one, consider the MF and the tWF coefficients. From Figure 9.3, it is evident that MF coefficients tend to be closer to 1 than the tWF ones. Even when they are not very good, such as the case of `gzip` v2, they are greater than the tWF coefficients. The difference is big for three out of the 7 cases and always greater to all the tWF coefficients. This denotes a higher correlation between the mutants and the faults than between the t-wise coverage and the faults.

For instance, consider the results for `flex` v1, depicted by Figure 9.3a. The maximum coefficient τ is very close to 1. The median τ among the 30 coefficients is above 0.87. It Thus denotes a strong correlation for the MF case. Regarding the t-WF results, median coefficients are below 0.5, which denote moderate correlations between t-wise and the faults found by the test cases. Conclusively, it can be argued that the comparison is in favor of the MF since it gives correlations greater than the t-WF.

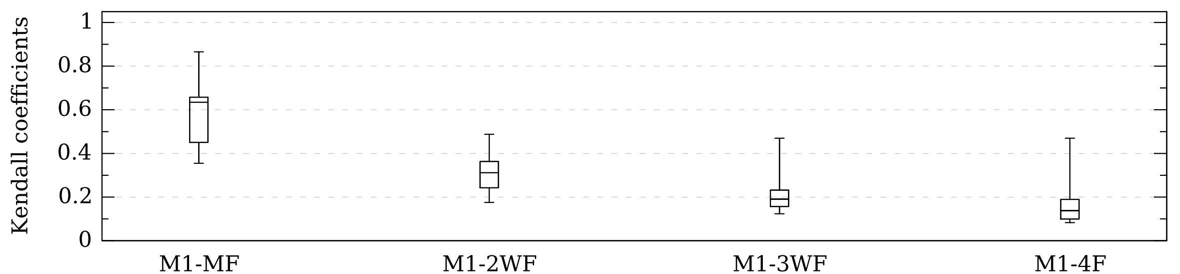
The results presented so far consider the correlation of CIT and mutation with the code-based faults. However, this correlation may be missleaded by the difficulty of finding the faults. In other words, if



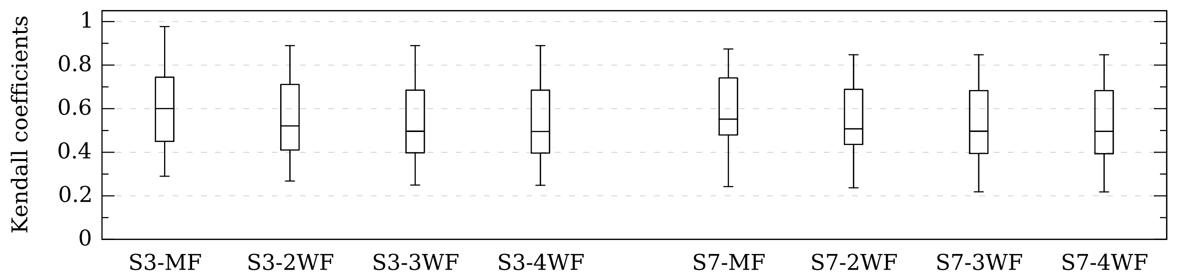
(a) Results for `flex` v1 and v4. FX denotes `flex` vX, MF denotes the correlation mutants killed and faults found, and tWF denotes the correlation t-wise coverage and faults found.



(b) Results for `gzip` v1 and v2. GX denotes `gzip` vX, MF denotes the correlation mutants killed and faults found, and tWF denotes the correlation t-wise coverage and faults found.



(c) Results for `make` v1. MX denotes `make` vX, MF denotes the correlation mutants killed and faults found, and tWF denotes the correlation t-wise coverage and faults found.



(d) Results for `sed` v1 and v7. FX denotes `sed` vX, MF denotes the correlation mutants killed and faults found, and tWF denotes the correlation t-wise coverage and faults found.

Figure 9.3: Distribution of the Kendall τ rank coefficients on different versions of the subject programs. Each boxplot represents the distribution of the 30 τ coefficients of correlation between either the mutants killed and the faults found or between the t-wise ($t = 2, ..., 4$) coverage and the faults found.

the faults are very easy to find or very difficult to find, this will have a direct effect on the measured correlations. In the same lines, we can compare the CIT and the mutation approaches.

Figure 9.4 presents the easiness of finding faults, killing mutants, covering 2-wise and covering 3-wise interactions per subject. The easiness of finding a fault represents the percentage of test cases that find this fault. Similarly, the easiness of killing a mutant represents the percentage of test cases that kill this mutant, and the easiness of covering a t-wise interaction is the percentage of test cases that cover this interaction. FX, GX, MX, and SX respectively represents the `flex`, `gzip`, `make` and `sed` subjects, with the corresponding X version. For each subject, the fault box represents the easiness for faults, the mutants box represent the easiness of the mutants and the t-wise box the easiness of the t-wise interactions.

From these results, we can explain why both CIT and mutation have low correlation values for `gzip` v2. This is due to the fact that the faults of this version are very easy to detect, contrary to mutants and interactions. It is the same for `make`. However, in this case, both mutants and interactions are easy to detect, resulting in satisfactory correlations.

Conclusively, from the presented results, it becomes evident that killing mutants is more or less as difficult as covering 2-wise and 3-wise interactions. Indeed, the easiness median values for mutants are comparable with the easiness medians of 2-wise and 3-wise interactions for `sed`, greater for `gzip` and lower for `flex` and `make`. Considering the easiness of mutants and faults, we observe that mutants tend to behave similarly as faults.

Finally, covering 3-wise interactions is harder than covering the 2-wise ones. This is expected since covering all 3-wise interactions results in covering all the 2-wise ones. As a result, higher interactions strengths ($t \geq 4$) are more difficult to cover.

9.6 Discussion

The findings of the conducted experiment suggest that the mutation approach can form an alternative method to CIT. Based on the results, we can infer that mutation is probably more effective in predicting the actual fault detection of a test suite. In view of this, some additional considerations regarding the comparison with CIT and the application cost of the approach are needed. This section discusses these considerations as long as threats to the validity of the conducted experiment.

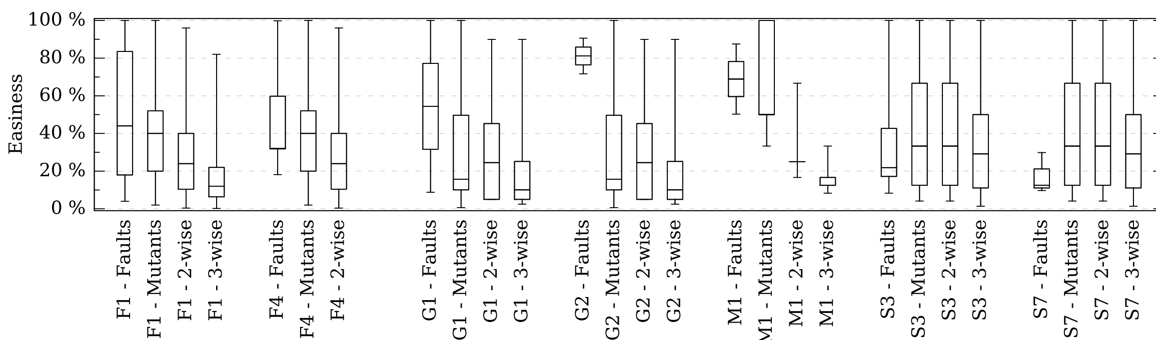


Figure 9.4: The easiness of finding faults, killing mutants and covering t-wise interactions per subject. The easiness of finding a fault represents the percentage of test cases that find this fault. Similarly, the easiness of killing a mutant represents the percentage of test cases that kill this mutant, and the easiness of covering a t-wise interaction is the percentage of test cases that cover this interaction. FX, GX, MX, and SX respectively represents the `flex`, `gzip`, `make` and `sed` subjects, with the corresponding X version. For each subject, the fault box represents the easiness of the faults, the mutants box represent the easiness of the mutants and the t-wise box the easiness of the t-wise interactions.

9.6.1 Additional consideration about mutation and combinatorial testing

By considering the CIT and the mutation approaches, we can observe that they more or less both use the same information, which is the input model. However, they provide much different results. Why this difference? In other words, why mutants can be more powerful than the input combinations?

Generally, it is very hard to fully answer this question. A full answer requires extensive and independent studies. However, we believe that the power of mutation lies on the fact that it considers the input constraints of the tested systems. Recall that our benchmark programs are real word programs. Thus, they have input constraints. These constraints play an important role in the testing process of the system. Not only they define the valid program inputs but they also reflect a logic of the underlying system. Therefore, they provide some useful information to the testers which is actually missed by CIT. Mutation takes advantage of this information by mutating these constraints. Mutating the input constraints forces tests to exercise limit cases that trigger faults. From the testing perspective, input constraints provide information similar to the one provided by boundary conditions of a system. They also have a role which is similar to the role of the preconditions of a system. Testing preconditions and boundary conditions of a system has been identified as an important step of the testing process [FZ11]. Hence, testing them is necessary for establishing a rigorous testing approach. Here, it should be noted that CIT uses the input constraints only for computing the valid combinations of program inputs. Thus, it completely ignores both their importance and the information they provide.

Generally, mutation analysis relies on the power of the utilized mutants. The present studies uses two Boolean operators mainly chosen based on the authors experience from the feature modeling of software product lines [HPP+13a, HPP+13b, HPP+14]. Feature models are Boolean models like the flattened ones used by the present study and hence, the employed operators form a good choice. In future, we plan to investigate the use of other operators, e.g., [KPAO11] and higher order ones [JH09, KPAO11]. Nevertheless, the performed analysis shows that the utilized mutants simulate very well the behavior of the actual faults. Here, it should be mentioned that the use of mutant selection strategies can increase the difficulty of finding them. However, the main question is whether doing so results in accurate estimations of the actual fault detection. Particularly, we do not want to underestimate or overestimate our measures [ABLN06]. This matter falls outside the scope of this chapter since it is a general research challenge of the whole mutation testing area.

9.6.2 Cost of the approach

One of the main issues of mutation analysis is its computational cost [JH11]. This is due to the need to introduce and execute a vast number of mutants. However, in our context the computational cost of mutation analysis is not very important due to the following three reasons. *First*, we only check whether a mutant violates the Boolean constraints of the mutated models. This is a simple SAT verification process which is actually quite fast. *Second*, the number of mutants is small. Actually much smaller than the number of interactions, see Table 9.1. For 2-wise, the number of mutants is 5 times lower than the number of interactions. As a result, mutation requires less operations than CIT. *Third*, we do not execute the system. Test selection and evaluation based on the input model is much faster than the actual program execution. Furthermore, in practice, testers will have to verify the program behavior, i.e., resolving the oracle problem, which is a typical manual activity. Hence, human time dominates the computational expenses of the approach.

Finally, it should be mentioned that equivalent mutants, i.e., mutants that can not be killed by any test case do not introduce a big overhead. Actually, the number of such mutants is much less than

the number of invalid pairs (Table 9.1). Therefore, the computational cost of mutation is lower than the cost of CIT.

9.6.3 Threats to validity

There are several influencing factors that can threaten the validity of the conducted experiment. Regarding the generalization of the findings, i.e., *external validity*, it is possible that the selected programs are not representative. This may also be the case for the utilized test suites and the faulty versions. Thus, on larger or other types of programs, mutants and input interactions might not be the most appropriate choice. Similarly, the examined approaches might not being effective in revealing other faults. However, the chosen subjects are real world programs widely used in the literature e.g. [PYCH13, PLT13, YHC13]. Additionally, both the test suites and faults were developed by researchers independently of the present study. The problem we are facing is the absence of programs with high quality test cases, well defined input models or specifications, and faulty versions. Clearly, more studies are in need to answer this concern with confidence.

With respect to the confidence on the reported results, i.e., *internal validity*, issues on the utilized input models, the employed test sets and the correctness of the used tools can be identified. It is possible that errors on the input models and the used tools may have influenced the reported results. To reduce this threat we performed several manual checks on both the implementation and the employed input models. Here, it must be noted that the input models were independently developed by other researchers [PYCH13]. They were also checked by us in order to give confidence about their correctness. Additionally, as already mentioned, the employed suites are widely used in software engineering experiments e.g. [PYCH13, PLT13, YHC13].

Finally, some threats regarding the evaluation metrics used, i.e., *construct validity*, can be identified. It is likely that the number of faults found by the approaches do not express the real fault detection ability of the test suites. Additionally, it is possible that the faults number and difficulty can influence the significance of the performed statistical analysis. To reduce this threat, we employed the Kendall τ coefficient which is a non-parametric hypothesis test, i.e,. it does not require a very big sample size, and it measures the similarity of the data order when ranked by the studied effectiveness measures, i.e., the mutants found or the interactions covered. We also measured the correlations after executing every test case of various test suite sizes to eliminate the effects of the test suite size.

9.7 Conclusions

In this chapter, we proposed a mutation analysis approach as an alternative technique to CIT. We conducted a correlation analysis between a) the CIT and b) the mutation approach with their actual fault detection. Our results suggest that our mutants have a stronger correlation with code-level faults than the input interactions of the CIT approach. Therefore, mutation forms a valid measure of the test suites quality.

Part V

Reverse-engineering and re-engineering

10

From software product variants to a software product line: a preliminary approach

In the previous parts, we presented methods for generating and evaluating configurations. However, a software product line is not always available. The goal of this chapter is to present a preliminary but fully automated approach for reverse-engineering software product lines and their feature model from the source code of software product variants.

This chapter is based on the work that has been published in the following paper:

- Tewfik Ziadi, Christopher Henard, Mike Papadakis, Mikal Ziane, and Yves Le Traon. Towards a language-independent approach for reverse-engineering of software product lines. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1064–1071, New York, NY, USA, 2014. ACM

Contents

10.1 Introduction **128**
10.2 The ExtractorPL approach **128**
10.2.1 The banking system example 129
10.2.2 Abstraction of the software product variants 129
10.2.3 Automatic identification of the software product line features 131
10.2.4 Feature code generation 133
10.3 Case study **133**
10.3.1 Accuracy of the extracted software product line (research question 1) 134
10.3.2 Quality of the extracted software product line (research question 2) 135
10.3.3 Threats to validity 137
10.4 Discussion **137**
10.4.1 Benefits 137
10.4.2 Limitations 138
10.5 Conclusions **138**

10.1 Introduction

SPLE can be implemented as a top-down approach. In that case, features and variability are first specified at the design stage and then software products are built. The top-down process is useful when SPLE is adopted from the beginning. However, current practices are different. Indeed, as reported by Berger *et al.* [BRN+13], most of the industrial practitioners first implement several software products and then try to manage them. These SPVs are created using ad-hoc techniques, e.g., copy-paste-modify. This is a bad practice leading to a complex management and a low SP quality. Thus, migrating such SPVs to a SPL is the challenge faced by extractive approaches in SPLE [Kru02].

The need for reverse-engineering the full implementation of a SPL. Over the past decade, several studies have investigated such approaches. Some of them deal only with the extraction of features from textual requirements [ASB+08], architectural artifacts [ACC+14, KK12, RPK10], or SPV descriptions [ACP+12]. We advocate that these techniques can go beyond the feature identification step. Indeed, they can refactor and migrate SPVs into a SPL. In other words, a full implementation of a SPL can be reverse-engineered. Therefore, not only features but also their associated assets, e.g, code units, should be identified and extracted.

Contributions of this chapter. This chapter introduces an automated technique, called *ExtractorPL*, capable of performing a reverse-engineering of a SPL. ExtractorPL infers a full implementation of a SPL given the source code of SPVs. The main challenge of this task is to analyze the source code of the SPVs in order to:

1. Identify the variability among the SPVs,
2. Associate them with features,
3. Regroup the features into a variability model,
4. Map the code units to each feature.

The proposed approach is language-independent and only uses as input the source code of the SPVs. In addition, ExtractorPL is implemented as a publicly available prototype tool and a case study performed on existing SPLs assesses the feasibility and the practicality of the introduced technique.

The remainder of this chapter is organized as follows. Section 10.2 and 10.3 respectively present the ExtractorPL approach and a case study to evaluate it. Section 10.4 discusses the benefits and the limitations of the approach. Finally, Section 10.5 concludes the chapter and present our future work.

10.2 The ExtractorPL approach

ExtractorPL is a language-independent approach which extracts a SPL from the source code of SPVs. The reverse-engineered SPL is a full implementation since it allows (a) building specific products by composing the source code units of the identified features and (b) managing the resulting SPL and its products through a FM.

To achieve this re-engineering, three main steps are considered. These steps are depicted by Figure 10.1. First, ExtractorPL abstracts each SPV into a set of atomic pieces called set of construction

primitives (SoCPs). Each construction primitive (CP) represents a node in an abstract syntax tree for features called feature structure tree (FST). Then, following the lines suggested in [ZFdSZ12], features are identified as SoCPs and translated to FSTs. Finally, the code units are generated from the obtained features' FSTs.

The following subsections introduce an example of SPVs from a banking system that can be migrated into a SPL. Then, the three main steps of the ExtractorPL approach are detailed through the example.

10.2.1 The banking system example

As a concrete example of SPVs, consider a set of banking systems [ZFdSZ12]. Each variant proposes a simple banking application. The variability between these SPVs is related to the limit on the account and to the currency exchange, which are optional features. These banking products were manually developed using a copy-paste-modify approach.

Figure 10.2 illustrates the source code of the `Account` class in the three variants which are denoted as `Product1Bank`, `Product2Bank`, and `Product3Bank`. Following this figure, consider the `Product1Bank` product depicted by Figure 10.2a. In this SP, the `Account` class defines a basic banking account without the limit and currency information. On the contrary, since the `Product2Bank` of Figure 10.2b supports an account limit feature, its corresponding `Account` class defines the `limit` field. In addition, the `withdraw` method is refined to check the limit. Finally, in the `Product3Bank` variant of Figure 10.2c, the `Account` class is defined with information related to both the limit and currency exchange.

10.2.2 Abstraction of the software product variants

ExtractorPL takes as input the source code of a set of SPVs. To analyze and compare these variants, each SP is first abstracted into a SoCPs. To this end, ExtractorPL builds the SoCPs associated to each SP by using the general model proposed within FeatureHouse, called Feature Structure Tree

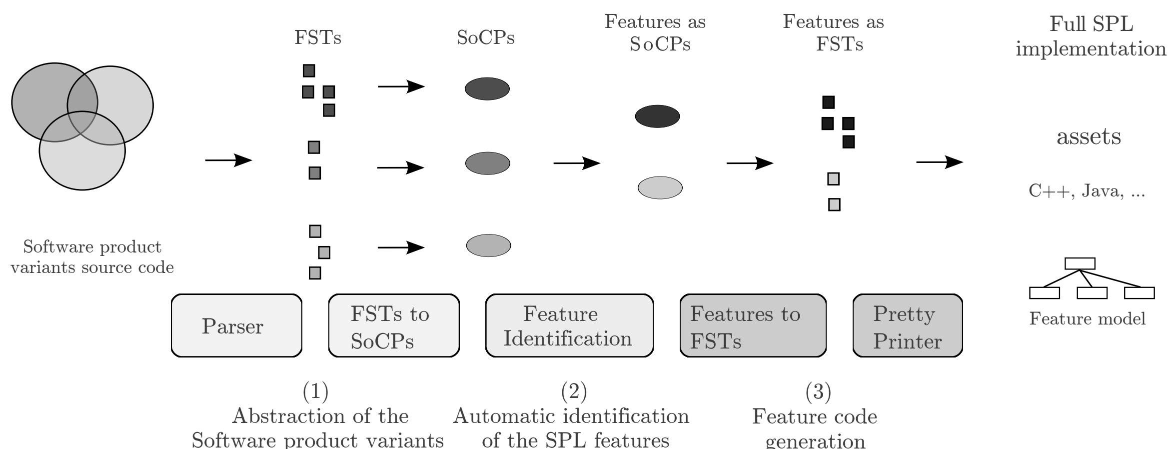


Figure 10.1: The ExtractorPL approach for the reverse-engineering of software product lines. First, the software product variants are abstracted. Then, their features are automatically identified. Finally, the code units corresponding to the features are generated and a feature model is extracted.

(FST) [AKL09]. a FST represents the essential software artifacts as a tree. Each node of a FST has a name and a type.

The choice of FSTs is based on the following two criteria. First, a FST is a language-independent model. Second, FSTs include general composition operators. From the FSTs associated to each SPV, ExtractorPL represents them as a SoCPs. The construction primitives (CPs) used to decompose each FST depend on the type of nodes within each FST. In this work, the following construction primitives are used:

SoCPs = {`CreateNonTerminal(name, type, child)`, `CreateTerminal(name, type, parent, body)`}.

We distinguish two types of nodes within a FST: *non-terminal* and *terminal* ones. A non-terminal node denotes inner modules, e.g., packages and classes. Terminal nodes store the module's content, e.g., method bodies [AKL09].

Each product variant is thus abstracted as a set of FSTs. For each node in each obtained FST, a CP is created and added to the SoCPs. This means that each SP is defined as a set $P_i = \{cp_1, cp_2, .., cp_n\}$, where each $cp_i \in$ SoCPs. In the following, we consider $AllP = \{P_1, P_2, .., P_N\}$ as the set of SPVs available to perform the reverse-engineering of the SPL, i.e., the input of the ExtractorPL approach.

The left part of Figure 10.3 depicts the three FSTs obtained from the source code of the banking SPVs. For instance, the `Account` class is represented by a node with the name `Account` and the type `Class`.

The following subsection introduces the algorithm that compares and analyzes the extracted SoCPs. Equivalence between the construction primitives relies on the equivalence between nodes in the corresponding FST [AKL09], as defined below.

```
package bs;
public   class   Account {
      private String id;
      private double balance;
      public void deposit(double amount) {
            this.balance += amount;
      }
      public void withdraw(double amount) {
            if (amount <= balance)
                  balance -= amount;
      }
}
```

(a) The `Product1Bank` variant

```
package bs;
public     class    Account {
      private String id;
      private double balance;
      private double limit;
      public void deposit(double amount)    {
            this.balance += amount;
      }
      public void withdraw  (double amount) {
      if (amount <= balance + limit)
            balance -= amount;
      }
      public double getLimit() {
            return limit;
      }
}
```

(b) The `Product2Bank` variant

```
package bs;
public     class   Account {
      private String id;
      private double balance;
      private double limit;
      private double currency;
      public void deposit(double amount)    {
            this.balance += amount;
      }
      public void withdraw  (double amount) {
      if (amount <= balance + limit)
            balance -= amount;
      }
      public double getLimit() {
            return limit;
      }
}
```

(c) The `Product3Bank` variant

Figure 10.2: Example of three software product variants of a banking system. Each variant has a specific implementation of the `Account` class.

1. A non-terminal node n_1 is equivalent to a non-terminal node n_2 if and only if n_1 and n_2 have the same name, the same type and the same node child.
2. A terminal node n_1 is equivalent to a terminal child n_2 if and only if they have the same name, the same type, the same parent and the same body.

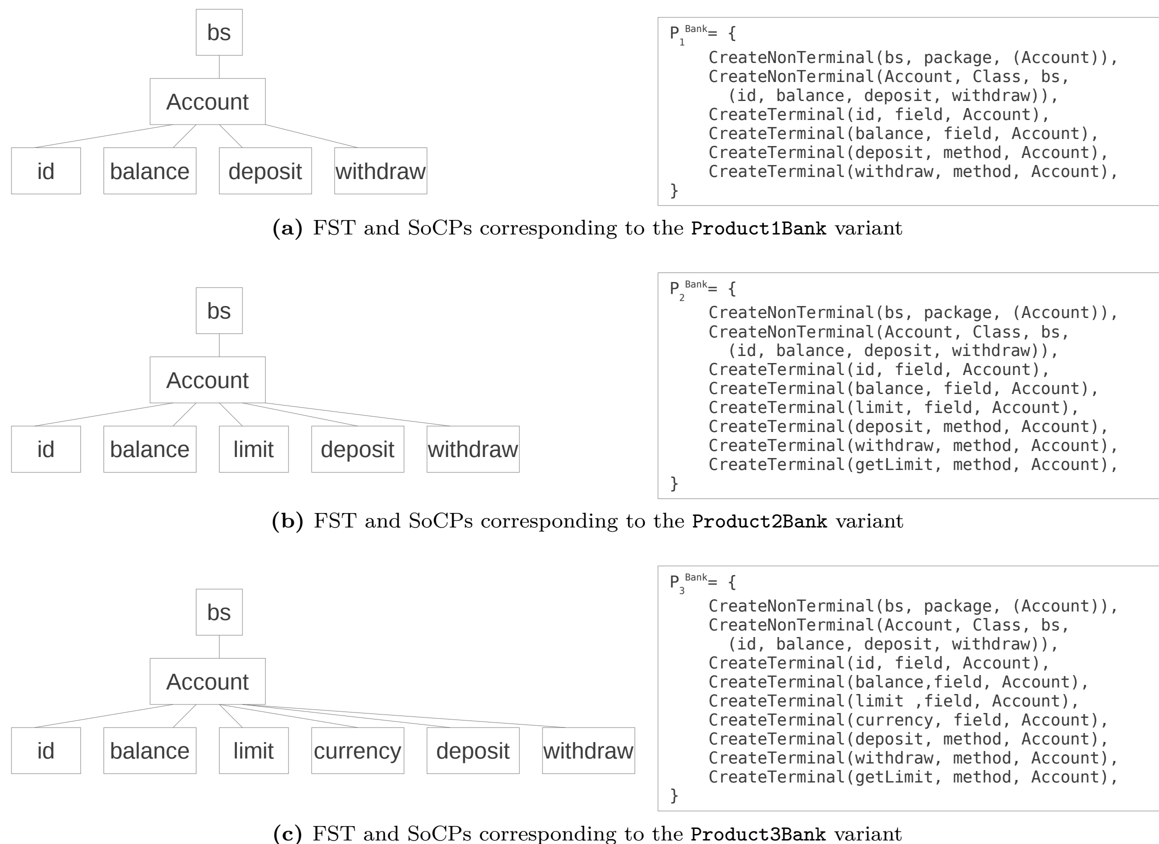
10.2.3 Automatic identification of the software product line features

The second step of the reverse-engineering process performed by our approach aims at comparing the SoCPs of the SPVs in order to identify their features. To this end, ExtractorPL first uses the algorithm proposed in [ZFdSZ12] to represent features as sets of SoCPs. Then, these features are transformed into FSTs.

The feature identification process is based on a formal definition of a feature that uses the notion of interdependent CPs. This notion is defined as follows.

Definition 8 (Interdependent construction primitives) *Given the set of SPVs that can be used by ExtractorPL, AllP, two CPs (of SPs of AllP) cp_1 and cp_2 are interdependent if and only they belong to exactly the same SPVs of AllP. In other words, cp_1 and cp_2 are interdependent if the two following conditions are fulfilled.*

1. $\exists P \in AllP \;\; cp_1 \in P \wedge cp_2 \in P$.



(a) FST and SoCPs corresponding to the `Product1Bank` variant

(b) FST and SoCPs corresponding to the `Product2Bank` variant

(c) FST and SoCPs corresponding to the `Product3Bank` variant

Figure 10.3: Feature structure trees and construction primitives for the banking example. The left part represents the feature structure trees abstracting the `Account` class of the banking products variants. The corresponding set of construction primitives is represented on the right.

2. $\forall P \in AllP \;\; cp_1 \in P \Leftrightarrow cp_2 \in P$.

Since interdependence is an equivalence relation on the set of CPs of $AllP$, it leads us to the following definition of a feature.

Definition 9 (Feature) *Given $AllP$ a set of SPVs, a feature of $AllP$ is an equivalence class of the interdependence relation of the CPs of $AllP$.*

The application of this algorithm to the SoCPs of the banking SPVs provides the features depicted by Figure 10.4. This includes one mandatory feature and three optional ones. The `Base` feature gathers all the CPs that are present in all the SPVs. The feature `F1` concerns the limit information. Indeed, it contains primitives to create the `limit` field, its getter and the withdraw method with the body defining limit checking. `F2` is related to the currency exchange since it contains the CPs related to the currency field. The `F3` feature is related to the withdraw method without limit checking. Finally, ExtractorPL also organizes the obtained features into a FM. This model is depicted by Figure 10.5.

The algorithm for identifying the features of a given set of products [ZFdSZ12] is based on this definition. In brief, it takes as input a set of SoCPs, i.e., one per input SPV and returns a single mandatory feature called `Base` and a set of optional features for these SPVs. Finally, once the features of the SPVs have been identified, they are represented as FSTs in order to be useful for generating the code of each feature.

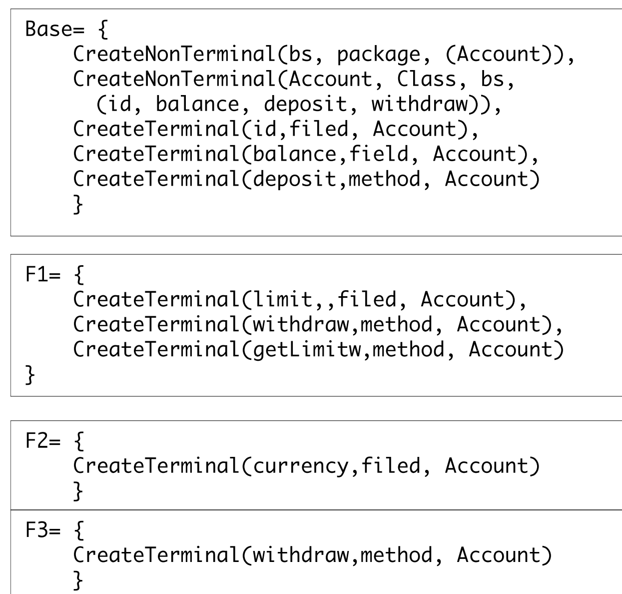


Figure 10.4: The identified features for the banking software product variants. `Base` gathers the code common to any banking system. `F1` concerns the limit information. `F2` represents the currency exchange and `F3` denotes the withdraw method without the account limit checking.

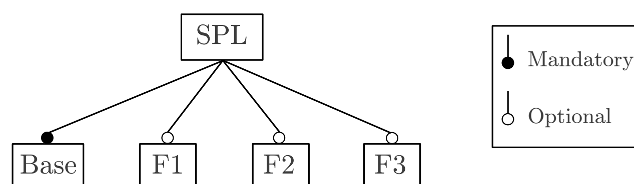


Figure 10.5: The feature model built by ExtractorPL for the banking example.

10.2.4 Feature code generation

In addition to the feature identification, ExtractorPL aims at extracting a full compositional implementation of a SPL from the source code of SPVs. It means that our approach includes the generation of the code units associated to each feature once the SPL is extracted. As a result, one is able to build tailored software products by automatically generating the source code of the features selected.

To perform the code generation, ExtractorPL uses the FSTs of the features obtained in the previous step. Then, it generates their code units by using the pretty printers proposed by FeatureHouse. Indeed, the FeatureHouse framework includes a set of pretty printers for different programming languages such as C or Java. These printers produce the code from FSTs. Using this framework within our approach allows ExtractorPL being a language-independent technique.

Finally, to demonstrate the result of the code generation, Figure 10.6 depicts the code units generated from the FSTs of the features. Following this figure, the code associated with the `Base` feature represents the common implementation of the `Account` class. The code units of the remaining optional features express the variability and refine the common code by adding elements related to each feature. Given the code of each feature, a product variant of the resulting SPL can be built using the compositional operators between FSTs. The following section evaluates ExtractorPL via a case study.

10.3 Case study

In this section, ExtractorPL is assessed. The question raised here is how this evaluation can be performed. Generally, an automated reverse-engineering task can be considered as successful when it provides similar results to a manually performed one. Therefore, one way to measure the accuracy of our approach is to compare its result with the result from a developer. Another way is to measure

```
package bs;
public   class   Account {
     private String id;
     private double balance;
     public void deposit(double amount) {
          this.balance += amount;
     }
}
```

(a) The code generated for the `Base` feature

```
package bs;
public     class    Account {

     private double limit;

     public void withdraw  (double amount) {
     if (amount <= balance + limit)
          balance -= amount;
     }
     public double getLimit() {
          return limit;
     }
}
```

(b) The code generated for the `F1` feature

```
package bs;
public     class    Account {

     private double currency;

}
```

(c) The code generated for the `F2` feature

```
package bs;
public     class    Account {

     public void withdraw(double amount) {
          if (amount <= balance)
               balance -= amount;
     }
}
```

(d) The code generated for the `F3` feature

Figure 10.6: The code units generated for each of the extracted features of the banking example.

Table 10.1: The notepad software product line used to evaluate ExtractorPL towards research question 1.

Lines of code	806
Classes	14
Optional features	Copy/Cut/Paste (CCP) Undo/Redo (UR) Find (F)
Product variants	Basic Notepad (BN) BN + CCP BN + UR BN + F BN + CCP + UR BN + CCP + F BN + UR + F BN + CCP + UR + F

whether the extracted SPLs provide a minimum quality standard as defined by the two following requirements.

1. The extracted SPL allows building the SPVs that have been used to perform the re-engineering.
2. The approach identifies the features that must appear in all the possible variants of the SPL. These features are usually called mandatory features.

If a reverse-engineered SPL cannot fulfill the first condition, it is obviously an erroneous approach. Similarly, if the second condition cannot be satisfied, then there is no hope to identify optional features. Following the above-mentioned concerns, a controlled experiment is conducted based on existing SPLs. We use SPVs from existing SPLs in order to establish a comparison basis between the existing SPLs and the extracted ones. As a result, this case study aims at answering the two following RQs:

- [RQ1] *How close the SPL reverse-engineered by ExtractorPL is to the original one?*
- [RQ2] *Does the reverse-engineered SPL has a minimum quality standard?*

The first RQ amounts to evaluate whether the SPL extracted with ExtractorPL conforms to the original one. To this end, we manually compare the original SPL with the extracted one. In particular, we check whether extracted features correspond to those defined in the original version of the SPL. The second RQ aims at checking whether the above-mentioned minimum requirements are fulfilled by the SPL extracted by our approach.

The evaluation of ExtractorPL is divided into two parts. The first one is performed manually and aims at answering to the *RQ1*. The second one performs automatically in order to answer to the *RQ2*.

10.3.1 Accuracy of the extracted software product line (research question 1)

In this section, we compare the SPL resulting from ExtractorPL with the original one. This is a manual step and thus it requires a lot of effort to be accomplished. Therefore, in order to complete the experiments with reasonable resources, the evaluation is limited to one benchmark.

10.3.1.1 Setup

We use a notepad SPL written in Java [TKB+14]. This SPL is detailed in Table 10.1. It contains 14 Java classes for a total of 806 lines of code. It proposes three optional features: copy/cut/paste, undo/redo and find. By combining these three features, up to 8 different notepad applications can be built. These 8 SPVs are used by ExtractorPL to reverse-engineer a SPL. We manually compare the features extracted by ExtractorPL using the 8 SPVs with the features of the notepad SPL.

10.3.1.2 Evaluation

ExtractorPL has extracted 4 features. The first one, `Base`, contains the 7 classes related to the core of any notepad variant: `About`, `Actions`, `Center`, `ExampleFileFilter`, `Fonts`, `Notepad` and `Print`. The three other features, `F1`, `F2`, and `F3` are related to the optional features of the original SPL. Indeed, `F1` contains the `Notepad` and `Actions` classes. This latter defines the copy, cut and paste methods. As a result, `F1` is related to the copy/cut/paste feature. The second feature, `F2`, encompasses the `Notepad` and `Actions` with the find method and attributes. Finally, `F3` contains the `Notepad, Redo` and `Undo` actions. Finally, we manually checked that the extracted SPL allows generating the code of all the SPVs used as input and that they can be executed without encountering any problem.

10.3.1.3 Answering research question 1

The extracted SPL conforms with the original one. We compared the obtained features and variants and found that our approach is able to accurately retrieve the features and to re-generate the SPVs used as input. As a result, given a set of SPVs ExtractorPL is able (a) to retrieve the variability among these SPVs and (b) to extract a SPL that is representative of the original one.

10.3.2 Quality of the extracted software product line (research question 2)

The second part of this study aims at evaluating whether the reverse-engineered SPL achieves a minimum level of quality.

10.3.2.1 Setup

We use the two following SPLs from FeatureIDE [TKB+14]:

1. E-Mail. This SPL gathers a family of systems managing mails. Examples of optional features in this SPL include encryption or the address book.
2. GPL. The Graph SPL is a family of graph manipulation algorithms [LHB01].

Table 10.2: The software product lines used for the evaluation of the approach towards research question 2.

	Language	Features	SPVs	Classes/Files	Lines of code	SPVs used by ExtractorPL
E-Mail	C	23	5,632	39	816	1, 5, 10, 50 and 100
GPL	Java	38	840	55	1929	1, 5, 10, 50 and 100

We choose these particular SPLs as they are considered to be standard benchmarks. Table 10.2 gathers detail regarding these two SPLs. In particular, for each SPL, it presents the programming language in which it is implemented, the number of features of the corresponding FM, the number of possible SPVs that can be built according to the FM and the number of input SPVs that have been used by ExtractorPL to extract the SPL. Indeed, we used a sample of SPVs to extract the SPL since hundreds of SPVs can be build from these SPLs.

The configuration of the SPVs used as input by our approach are configurations randomly selected from the space of all the possible configurations that can be generated from the FM, as used in the previous parts of this dissertation. For each configuration randomly generated, we use FeatureHouse to construct the corresponding SPV. The resulting SPVs are then used by ExtractorPL to reverse-engineer the SPL.

For each SPL and for each number of SPVs used by our approach, the extraction of the SPL has been independently performed 10 times. In the following, we present two approaches to automatically evaluate the resulting SPL. The first one compares the SPVs generated by ExtractorPL with the ones used to extract the SPL. We expect the resulting SPL to be able to build the SPVs that were used as input. The second steps automatically evaluates the mandatory features. Here, we expect the mandatory features of the original SPL to be included in the mandatory features of the extracted SPL.

10.3.2.2 Regeneration of the input software product variants

The objective is to check whether the extracted SPL allows building the SPVs that were used by ExtractorPL. To this end, we check whether the SoCPs of a given input SPV (of the original SPL) matches the SoCPs of the corresponding SPV built from the reverse-engineered SPL. More formally, if $AllP_{in}$ denotes the set of N input SPVs used as input and if $AllP_{out}$ denotes the set of N SPVs generated from the extracted SPL, we check that:

$$(\forall P \in AllP_{in})(\exists P' \in AllP_{out}) \mid P = P',$$

where P and P' are SPVs represented as a SoCPs. The equivalence between two SPVs P and P' is defined as an equivalence between their SoCPs. For each of the two SPL of Table 10.2 and for each number of input SPVs, all the 10 runs of the approach produced a SPL which allows regenerating the input SPVs, thus validating the above-mentioned condition.

10.3.2.3 Evaluation of the mandatory features

ExtractorPL extracts one mandatory feature called `Base` and a set of optional features. In this section, we evaluate whether the mandatory features of the original SPL are included in the mandatory feature of the SPL extracted with ExtractorPL. To this end, we check whether the SoCPs of the original mandatory features are included in SoCPs of the mandatory feature of our extracted SPL. More formally, if $SoCPs_{in}$ denotes the SoCPs of the input mandatory features and if $SoCPs_{out}$ denotes the SoCPs of the extracted mandatory feature, we check that:

$$SoCPs_{in} \subseteq SoCPs_{out}.$$

The evaluation has been performed 10 times independently per SPL. For each SPL and for each number of random SPVs used as an input by the approach, we observed that the original mandatory features are included in the mandatory feature of the extracted SPL. It is noted that all the mandatory features are always validated on both the E-Mail (C) and GPL (Java) SPLs, fact which demonstrates the ability of ExtractorPL to retrieve the mandatory features.

10.3.2.4 Answering research question 2

From the results presented in the previous sections, we found that (a) the extracted SPL allows building the SPVs used to extract this SPL, and (b) all the mandatory features of the original SPL are included in our extracted SPL. It means that our approach does not miss any information, thus fulfilling the minimum quality requirements defined in the beginning of this section.

10.3.3 Threats to validity

The conducted study involves three existing SPLs. As a consequence, there is a threat regarding the generalization of the results. Indeed, using different SPLs might lead to different results. To both reduce this threat and to provide a good sample of applications, we used three SPLs considered as standard benchmarks. These three SPLs are of different size and programming languages. Other threats can be identified due to the employed evaluation metrics. In other words, there is a risk that the quality measures are irrelevant towards the “real” quality of a SPL. To reduce this threat, we evaluate the approach using manual and automatic metrics. Additional threats can be due to our implementation. Indeed, potential errors in it might affect the presented results. To overcome this issues, we divided our implementation into modules to minimize the potential errors. We also make the prototype tool publicly available. Finally, there is a threat regarding results that could happen by chance. To minimize the risks attributed to random effects, we repeated the experiments 10 times independently.

10.4 Discussion

This section first discusses the reasons to migrate existing SPVs to a SPL. In this respect, several profits are provided by our tool. Then, some limitations regarding the proposed approach are highlighted.

10.4.1 Benefits

Migrating a set of existing SPVs to a SPL can lead to the following profits. First, in can reduce the developments costs. Indeed, a SPL allows building tailored software products by combining the features. It thus allows reusing existing code within different SPs. This can be performed automatically and without adapting the code.

Second, it can bestow a faster time to market. A full implementation of a SPL as proposed by our approach easily allows building the SPVs by only selecting the desired features. The SPVs are then generated by composing the code of the selected features. This allows configuring easily the SPVs depending on the targeted market. It also greatly decreases the time to build these SPVs and enables a flexible productivity.

Another outcome is the higher quality in the SPs. Migrating existing SPVs to a SPL leads to a reduced risk to introduce errors when new SPs are created. Indeed, creating SPVs with ad-hoc techniques like copy-paste-modify can lead to an introduction of errors in some variants. If the code of each feature is centralized within the SPL and shared in all the SP proposing these features, it allows testing each feature independently. This allows using SPL testing techniques which aim at testing the whole SPL in an efficient way [HPP+13d, HPP+13c].

Finally, moving SPVs to a SPL provides a higher quality in the SPs developed, thus leading to an easier management and maintenance of the SPs. In particular, the variability model such as the FM allows managing and tailoring the SPs. Besides, the code contains less redundancy and is refactored according to the underlying model.

Regarding ExtractorPL, it is the first approach to the authors' knowledge which allows building a SPL from a set of SPVs. Indeed, existing approaches require additional information to perform the reverse-engineering, like annotations in the code. On the contrary, our approach is fully automated and requires only the source code of the SPVs. In addition to the extraction of the code units of the features, our approach also extracts a FM. Such a model provides a high-level view of the variability within the SPL. It also allows visualizing the features, their dependencies and paves the way to reasoning and model-based testing of the SPL [HPP+14].

10.4.2 Limitations

ExtractorPL does not consider variability within the body of methods or functions, i.e. the statement level. We are working on the extension of the approach to remove this limitation. The idea is to modify the FSTs grammar to allow defining nodes for the statements of functions. Besides, the current implementation of ExtractorPL only infers a FM with a single mandatory feature and a set of optional features. This model does not encompasses constraints among the features, e.g. implications or exclusions. We are working on an extension of the proposed approach to infer a possible list of constraints from the features that are observed in the SPVs. These constraints will then be proposed to the user to be accepted and added to the FM.

10.5 Conclusions

Automatically migrate a set of SPVs to a SPL is not an easy task. It requires to perform several non-trivial steps including (a) the identification of the features in the source code of the SPVs, (b) the extraction of the features as code units, (c) the extraction of a variability model and (d) once the SPL is extracted, the correct composition of these features in order to build tailored SPVs. We tackled this problem with ExtractorPL, a language-independent approach which provides a quick automatic front-end to refactor a set of similar SPVs into a SPL. ExtractorPL has been implemented in a prototype tool based on which several experiments have been conducted. Our technique bestows two main outcomes.

- **It is a full extractive approach.** From the source code of a set of SPVs, ExtractorPL extracts a full implementation of a SPL. This includes the features, their code units, and a variability model.
- **It is a language-independent approach.** ExtractorPL only manipulates FSTs to extract a SPL. To integrate new languages or artifacts, it only requires to implement a parser for FSTs related to this language.

Finally, to enable reproducibility of our results, our implementation of ExtractorPL is publicly available at `http://pagesperso-systeme.lip6.fr/Tewfik.Ziadi/sac14/`.

11

Fixing re-engineered software product line feature models

In the previous chapter, we presented an approach to perform the reverse-engineering of a software product lined and its feature model from the source code of software product variants. Since such approaches are generally error-prone, it is required to automatically fix non accurate feature models. In this context, this chapter introduces an approach to automatically test and fix re-engineered feature models so that they match the systems they model.

This chapter is based on the work that has been published in the following paper:

- Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Towards automated testing and fixing of re-engineered feature models. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 1245–1248. IEEE, 2013

Contents

11.1 Introduction 142
11.2 Test-and-fix loop 142
11.2.1 Testing the feature model 143
11.2.2 Fixing the feature model 144
11.2.3 Continuous improvement 144
11.3 Preliminary evaluation 145
11.3.1 The Linux kernel feature model 145
11.3.2 Evaluation of the re-engineered Linux kernel feature model 145
11.3.3 Improving the feature model 146
11.4 Conclusions 147

11.1 Introduction

To bring the benefits of feature modeling to HCS, several re-engineering techniques [SLB+11, LHGB+12], producing FMs from various systems artifacts, have been proposed.

The need for improving re-engineered feature models. These techniques are partly automated and often require human intervention. Such *re-engineered* FMs may thus not be accurate, yielding incorrect analyses decisions about highly configurable systems, in turn hampering their correct re-engineering. As an example of such inaccurate re-engineering, our experiments show that none of the 1,000 configurations generated from the Linux kernel FM [SLB+11] is consistent with respect to actual kernel configuration rules. This context motivates our two research questions:

- [RQ1] *How to detect inconsistencies between the re-engineered FM and its source highly configurable systems?*

Inconsistencies fall into two categories. On the one hand, system configurations derived from the FM are incorrect with respect to the system. On the other hand, existing valid configurations do not satisfy the FM formula. In this chapter, we refer as *testing* the process of finding these flaws. When detected, dealing with these discrepancies may require an automated correction of the FM.

- [RQ2] *How to automatically make a FM consistent with its real system?*

Digging manually through thousands of features and dealing with hundred thousands of possibly faulty constraints in a FM is not an option. Thus, one must devise automated ways to correct inconsistencies in the FM so that it reflects its system. The process of correcting a FM is referred to as *fixing*. Current re-engineering approaches either do not validate the re-engineered FMs or use simulation of the configuration process [SLB+11]. This practice, as our experiment shows, is insufficient to detect all the problems of the FM.

Contributions of this chapter. In this chapter, we propose an automated approach to both test and fix re-engineered FMs. It relies on a continuous loop where FMs are iteratively tested and fixed. This loop forms a search process that gradually improves (fixes) the FMs. The search is guided by the number of inconsistencies found during a continuous testing process. Early results on the Linux kernel FM shows that more than 50% of the problems encountered in the re-engineered FM can be eliminated.

In brief, this chapter provides the following insights:

- We introduce a search-based approach to automatically test and fix re-engineered FMs.
- We perform a preliminary evaluation to demonstrate the benefit of our approach.

The remainder of this chapter is organized as follows: Section 11.2 presents the test-and-fix loop. Section 11.3 reports on the empirical evaluation. Finally, Section 11.4 concludes the chapter.

11.2 Test-and-fix loop

Our approach involves two entities: the system and the re-engineered FM. To find and correct the problems of the FM, such as erroneous constraints, this approach requires two steps. The first one

aims at testing the FM to highlight the problems. The second step uses feedback information from the testing process to fix the FM. The repetition of these two steps form a test-and-fix loop illustrated by Figure 11.1.

11.2.1 Testing the feature model

Testing a FM consists of two parts: 1) the evaluation of the FM consistency using valid configurations of the system[i] and 2) the evaluation of configurations generated from the FM with respect to the system. This process is depicted by the upper box of Figure 11.1.

Evaluating the consistency of the feature model with respect to valid configurations of the system. This evaluation goes from the system to the FM. To this end, we assume the existence (or the possibility to obtain by some means) of working and actual configurations of the system. The FM is evaluated over these configurations to find existing constraints in the FM which are not compatible with the existing configurations. This first step gives feedback information on existing wrong constraints (EWC) in the FM. These EWC of the FM, as long as the existing system configurations that fail (SCF) to be validated through the FM, are returned.

Evaluating configurations generated from the feature model with respect to the system. This evaluation goes from the FM to the system. Valid configurations of the FM[ii] are randomly generated using a SAT solver [HPP+12]. These configurations are then evaluated on the system side. To this end, the tester decides whether the configurations are valid with respect to the system and provides feedback when configurations fail. We will consider that generally, the tester defines an oracle which uses abstract rules to decide upon the validity of configurations. The oracle depends on the system: for instance, it can use execution information or compilation result, and represents the task usually done by the tester. The oracle rules that fail (ORF) as long as the generated configurations that fail (GCF) according to the oracle are returned.

[i] A valid configuration refers to a working configuration of the system.
[ii] Here, a valid configuration is a configuration that satisfies the FM formula.

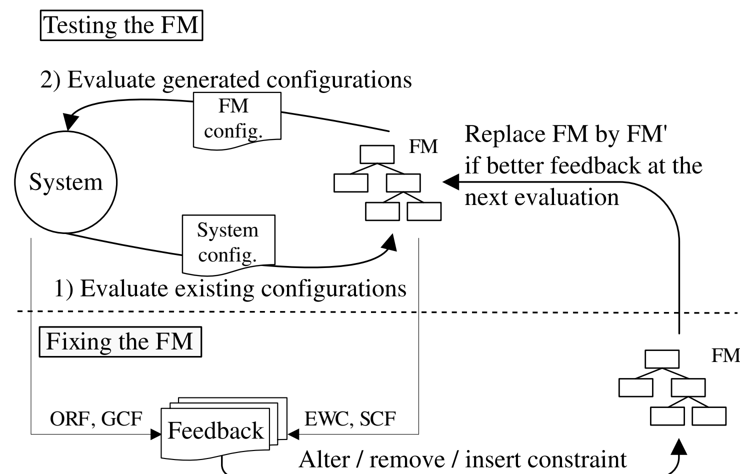


Figure 11.1: Test-and-fix loop for feature models. ORF are oracles rules that are violated on the system side, GCF are the configurations generated from the feature model (FM) that fail to be validated on the system side, EWC are existing wrong constraints in the FM and SCF are system configurations that do not satisfy the FM.

11.2.2 Fixing the feature model

This second step aims at fixing the FM based on the feedback information collecting during the testing part. This step is represented by the lower box of Figure 11.1.

To fix the FM, three operations are considered:

- *Altering* an existing constraint of the FM. Using the EWC, a constraint of the FM is selected based on a fitness proportionate selection and a randomly selected literal of the selected constraint is negated,
- *Removing* an existing constraint of the FM. Using the EWC, a constraint selected based on a fitness proportionate selection is removed,
- *Inserting* a constraint in the FM. Using the ORF, a constraint is added to the FM.

Performing one of this operation depends on a probability. After having performed one of this operation, an updated FM, FM', is produced. This resulting FM, FM', can be seen as a mutant of the original FM.

11.2.3 Continuous improvement

Basically, the global approach consists in the repetition of the testing and fixing steps. To this end, the problem is formulated as a search-based one and a hill climbing technique [RN03] is used. The approach works as follows. From a given FM, the testing step provides feedback information and the fixing part produces a modified FM FM'. Then, to decide which FM to keep, i.e. the original or the fixed one, the fitness of the original FM and the fitness of the fixed FM' are compared. This comparison is performed using the four feedback information of the testing step:

- s_1: the number of EWC or $\#EWC$,
- s_2: the number of SCF, or $\#SCF$,
- s_3: the number of ORF, or $\#ORF$,
- s_4: the number of GCF, or $\#GCF$.

Let us consider as $s_1, ..., s_4$ the feedback information of an FM and as $s'_1, ..., s'_4$ the feedback information of the fixed FM FM'. The updated FM FM' will replace the original FM if and only if a better fitness is observed for FM'. A better fitness for FM' occurs if the following condition is satisfied:

$$[(\sum_{i=1}^{4} s'_i < \sum_{i=1}^{4} s_i) \wedge (s'_1 \leq s_1 \wedge s'_2 \leq s_2 \wedge s'_3 \leq s_3 \wedge s'_4 \leq s_4)] \vee (s'_2 < s_2 \wedge s'_4 \leq s_4) \vee (s'_2 \leq s_2 \wedge s'_4 < s_4).$$

This condition allows ensuring that a decrease in one of the four feedback information does not engender any negative impact on the others and keeping the focus on reducing the configurations that fail. Finally, after having replaced or not FM by FM', the process is repeated from step (1). It should be noted that the loop is general and independent of both the way FMs are represented and the use of SAT solvers.

11.3 Preliminary evaluation

To evaluate the proposed approach, we consider the Linux kernel 2.6.28.6 FM[iii] [SLB+11].

11.3.1 The Linux kernel feature model

The Linux kernel is an operating system written in C with about 6,000 features. The re-engineered FM of the Linux kernel contains about 200,000 constraints. In the context of this study, an oracle is needed on the system side to decide whether a given configuration derived from the FM is valid or not. In the context of this study, we use the *make* tool as the oracle. Alternatively, the user could decide himself (play the role of the oracle) about the validity of the configurations. *Make* is a tool that assist the compilation process of system sources. To this end, *make* check rules which are specified by dependencies between the features in *Kconfig* files. These files are placed in the source code directories of the system. We parsed these files to extract these dependencies and to transform them into CNF constraints. It represents around 8,000 constraints. Thus, for a given configuration generated from the FM, it is checked whether this configuration satisfy or not the oracle rules. If yes, the configuration is considered as a valid configuration of the system. Otherwise, it is considered as invalid. It is noted that the use of *make* as an oracle to test is specific to this study. The test-and-fix loop is applicable for any kind of oracle.

11.3.2 Evaluation of the re-engineered Linux kernel feature model

The re-engineered FM contains several problems that have been found while performing the testing process. Recall that the testing process allows evaluating the FM through the EWC, SCF, GCF and ORF feedback information.

First, problems occur when evaluating the re-engineered FM formula with respect to valid configurations of the system. An alternative option provided by *make* is the generation of valid configurations of the system. The re-engineered FM has been evaluated over 1,000 working system configurations produced by *make*. By evaluating the FM through these working configurations of the system, we found that 50 constraints in the FM were not satisfied and none of these 1,000 configurations were able to satisfy the FM. In addition, major issues were highlighted such as mandatory features in the FM which never appear in any valid configuration of the system.

Second, configurations generated from the FM do not satisfy the constraints checked by the *make* tool (the oracle rules). We found that for, any 1,000 configurations generated from the FM, more than 28% of these constraints were not satisfied and that all these 1,000 configurations generated from the FM were invalid for the system. These problems are summarized in the column “Re-engineered FM” of Table 11.1. The existence of these problems motivate the proposed approach.

[iii] `http://code.google.com/p/linux-variability-analysis-tools/source/browse/?repo=formulas`

Table 11.1: Evolution of the feature model problems over the repetitions (testing feedback). *EWC* are existing wrong constraints in the feature model (FM), *SCF* are system configurations that do not satisfy the FM, *ORF* are oracles rules that are violated on the system side and *GCF* are the configurations generated from the FM that fail to be validated on the system side.

	Re-engineered FM	Fixed FM			
		2,000 runs	3,000 runs	4,000 runs	5,000 runs
#*EWC*	50	46	43	41	39
#*SCF*	1,000	885	556	498	455
#*ORF*	2,468	1,646	1,395	1,236	1,084
#*GCF*	1,000	1,000	1,000	1,000	1,000

11.3.3 Improving the feature model

We executed our approach on the re-engineered FM. For all the testing and fixing steps, we used 3 valid configurations of the system. We could have use more valid configurations, but in a realistic situation, only a small number of working configurations should exist. These configurations were randomly generated using *make* and used for all the repetitions of the test-and-fix approach. For the generated configurations, 6 were generated at each repetition of the approach. The probabilities to execute one of the three operations were assigned as follows: 0.5 for the alteration, 0.4 for the insertion and 0.1 for the removal.

Using the testing process, we evaluated the fixed FM at different level of repetitions, as shown in Table 11.1. The feedback information were obtained using the same 1,000 valid system configurations as those used for the evaluation of the re-engineered FM and using 1,000 configurations generated from the FM.

11.3.3.1 Answering research question 1

The evaluation of the *EWC*, *SCF*, *GCF* and *ORF* of the re-engineered FM emphasizes the inconsistencies between this FM and the system. We believe that these simple metrics characterize inconsistencies that may exist between highly configurable systems, FMs representing their variability and oracles checking the legality of highly configurable systems configurations. Yet, more detailed inconsistency types may be needed to improve user feedback and drive the fixing process.

11.3.3.2 Answering research question 2

The proposed approach uses both existing valid configurations of the system and configurations generated from the FM. By using only 3 valid configurations of the system, the proposed approach allows reducing wrong constraints in the FM while making the FM satisfiable towards existing configurations of the system. After 5,000 repetitions of the process, the proposed approach dropped system configurations that fail from 1,000 to 455, and divided by more than the half the violated rules of the system.

11.4 Conclusions

Fixing a re-engineered FM to make it in consistent with its corresponding system is not an easy task. It requires a lot of efforts to first check existing constraints and then correct them. In practice, it represents a manual work difficult to realize. In this chapter, a test-and-fix loop to automatically improve re-engineered FMs was presented. This loop is implemented using a search-based technique, since the exploration space makes impossible the application of other unscalable approaches. The proposed approach tries to make the FM conform to the real system. The novelty of this approach is that it achieves to effectively automate the identification and correction of FMs inconsistencies. It is the first approach, to the authors knowledge, that actually employs and checks actual configurations on a real system while most re-engineering techniques do not face the generation of real system configurations. The preliminary study conducted on the Linux kernel FM provides promising results as it allows reducing the problems observed in the FM, thus improving the alignment of the model regarding the actual system.

Part VI

Industrial application and final remarks

12

Testing card authorization systems with CETREL: a real-world case study

This chapter presents an industrial application of the techniques presented in the previous parts.

Contents

12.1 Context . . . 152
12.1.1 Migrating to a new card authorization systems . . . 152
12.1.2 A difficult migration . . . 152
12.1.3 Improving the testing process . . . 153
12.2 Approach . . . 153
12.2.1 Modeling authorizations as a product line . . . 154
12.2.2 Application of configuration generation and evaluation approaches . . . 154
12.3 Preliminary results and future investigations . . . 155
12.3.1 Current results . . . 156
12.3.2 Work in progress and perspectives . . . 156
12.4 Conclusions . . . 157

12.1 Context

CETREL is a Luxembourgish company created in 1985 offering and managing a variety of products and services associated to credit cards and electronic payments. The company also operates as a technical intermediate between cardholders and banks for processing and managing credit card transactions, handling thousands of credit card **authorizations** per day. An authorization is a network message which allows or deny a credit card transaction. For instance, a credit card payment to a merchant or a withdrawal at an automated teller machine will trigger a credit card authorization for validating or not the payment. An authorization encompasses different parameters such as the amount of the transaction or the merchant identifier. These parameters along with the banking situation of the cardholder are used to authorize or deny the transaction.

12.1.1 Migrating to a new card authorization systems

The CETREL company is currently undertaking a migration of their card authorization system from a legacy application to a new one developed by an external consulting company. To test their new system, they are replaying daily traffic of authorizations on both systems and compare the two outputs. They expect the new card authorization system to behave similarly as the legacy one, the authorizations being the test cases. Figure 12.1 depicts the testing approach performed by CETREL. Despite the benefits provided by the deployment of the new system, the migration is introducing several issues.

12.1.2 A difficult migration

The current status in the real time environment is critical due to the difficulty to perform the migration: about 1 incident per month in the production card authorization system occurs. More specifically, each new setup of the system implemented into production causes problems due to potential defects it contains. Fixing those defects is essential since they can lead to the unavailability of the authorization system processing the authorizations. The consequences are multiple and severe, among which are the following: non-operated transactions resulting in money losses for banks and

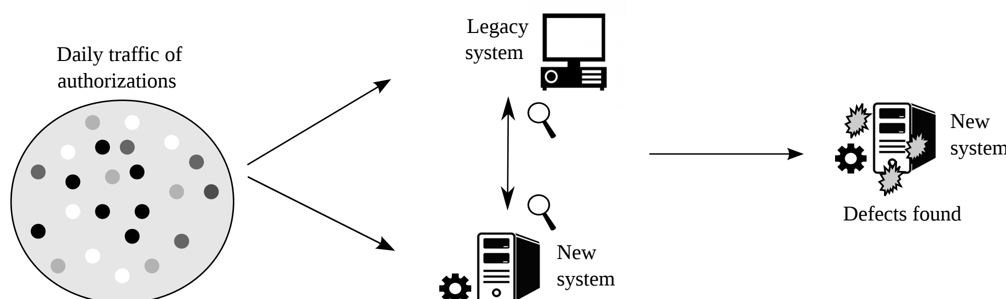


Figure 12.1: Process for testing the new card authorization systems in CETREL. Authorizations are played on both the legacy system and the new one. Different outputs or behavior indicate that the new system contains a defect.

merchants, damage in CETREL's reputation, and dissatisfied customers. Hence, fixing those defects is crucial.

12.1.3 Improving the testing process

Finding the defects due to the migration is essential, and require to improve the testing process. Replaying daily traffic of credit card authorizations to find potential defects in the new system is a costly process. For each authorization, it requires to set the banking context corresponding to the moment that authorization was issued in a test environment and to replay it. Since a daily traffic can contains dozen of thousands of authorizations, running all of them is a time consuming task.

Despite the time required for replaying daily traffics, some typical subsets of authorizations are used as standard test suites. However, there is no certainty regarding the quality of the authorizations that are used, i.e., their ability to find defects in the new system.

Thus, the objectives for improving the testing process are twofold: select a relevant subset of authorizations, i.e., **reduce a daily traffic of authorizations to a smaller subset**, and **evaluate the quality of a authorization set**. The first objective allows CETREL saving time from the testing process while the second one gives confidence in the test suites used. The following section describes the approach that was proposed to overcome these issues.

12.2 Approach

In a daily traffic, most of the authorizations are actually redundant. For instance, a scenario such as "a withdrawal of 20$ from an automated teller machine with sufficient money on the account" doesn't need to be tested multiple times. Thus, there is a degree of variability within a daily traffic of authorizations.

The idea we proposed is to consider authorizations as a product line. Thus, by modeling the variability of all the authorizations with a FM, authorizations represent configurations of this FM. It also enables the application of generation and evaluation, as described in Part II and Part III of this dissertation. The overview of the approach is depicted by Figure 12.2.

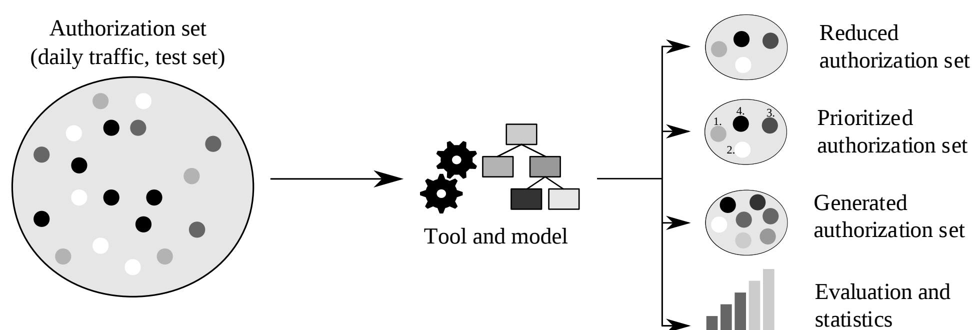


Figure 12.2: Approach proposed for improving the testing process prior testing. Using an authorization set as input and using the feature model representing the variability among the authorizations, our tool can a) reduce the authorizations by removing those that are redundant, prioritize the authorizations, generate new ones and evaluate them.

12.2.1 Modeling authorizations as a product line

Authorizations are messages which are structured by a header and parameters, as depicted in Figure 12.3. There are different types of headers, which are common to all authorizations. The bitmap fields indicates which parameters are set. The remainder fields correspond to the parameter. For instance, consider the following fragment of authorization[i] from a VISA credit card:

```
Message Type : VISR Emission date : Sat Jun 28 00:05:02 2014 length : 224
                                        Format header : 01
                                          Format text : 02
                                       Message length : 00E0
                                  Station Destination : 886103
                                       Station Source : 583011
                                   Round trip control : 04
                                         BASE 1 flags : 0000
                                 Message status flags : 060000
                                         Batch number : 00
                                         Reserve VISA : 00000B
                                            User info : 02
                                      Message Type ID : 0100
                                               Bitmap : F664648108F0A0160000000000000004
                                               P2 PAN : 4011887690553911
                                   P3 Processing Code : 000000
                               P4 Amount, Transaction : 000000006526
                        P6 Amount, Cardholder Billing : 000000006526
                              P7.1 Transmission Date : 0627
                              P7.2 Transmission Time : 230503
                                                  ...
```

It represents some of the parameters and their corresponding values. From such authorization messages, protocol specifications from the international card issuers such as Visa and Mastercard and experts advice from CETREL, we built a FM representing all the possible authorizations. The different authorization parameters represent the feature of this model. The resulting model encompasses more than 100 different parameters and about 150 constraints among them. Building such a model allows to apply the presented generation and evaluation techniques.

12.2.2 Application of configuration generation and evaluation approaches

Given the FM, we are able to:

[i]Some fields have voluntarily been hidden or change for preserving the confidentiality of the transaction.

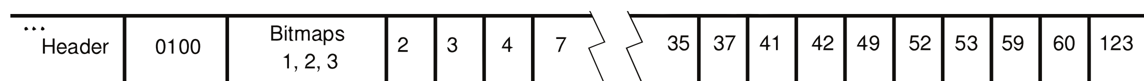


Figure 12.3: Structure of an authorization message. A header field is common to all the authorizations. The bitmaps field indicate which parameters are used. The remaining fields are numbered and correspond to the parameters of a credit card authorization, such as the amount of the transaction.

- Reduce the size of an authorization set. For instance, we can apply interaction coverage, e.g., 2-wise, to keep only authorizations covering interaction between two parameters.

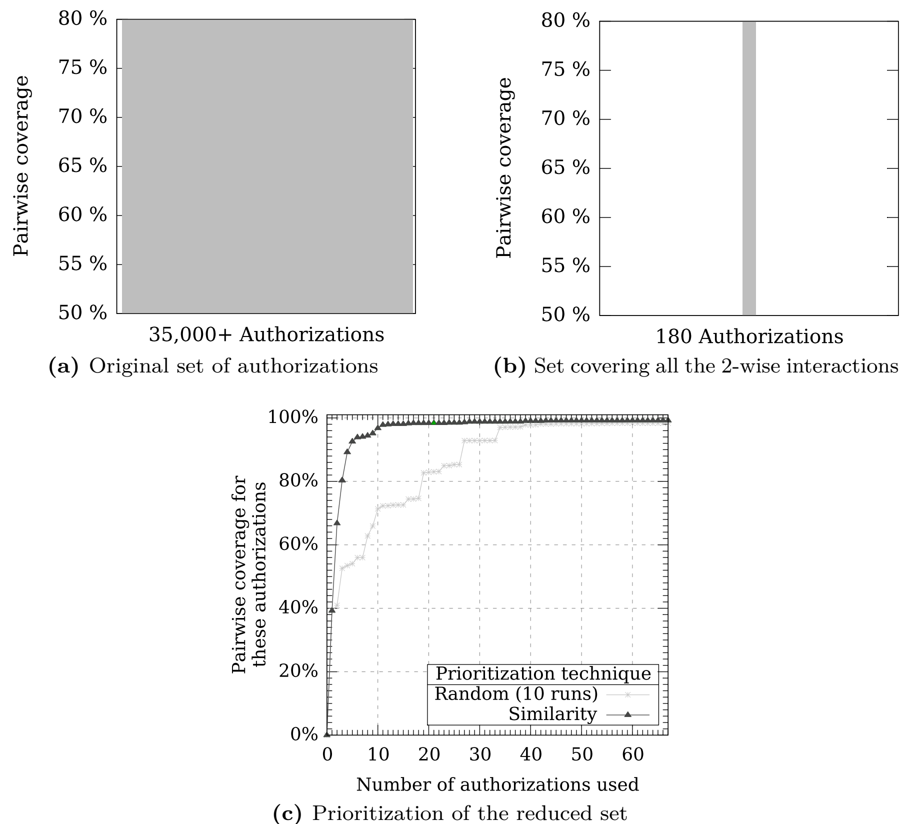
- Prioritize the authorizations. We can apply prioritization techniques such as those presented in Chapter 4. For instance, dissimilar authorizations can be tested first.

- Generate authorizations. Thanks to the FM, we can calculate all the possible authorizations. Thus, we can generate missing authorizations or parameters that are not covered in a given set using techniques presented in Part II

- Evaluate authorizations. For instance, we can assess the quality of authorizations by evaluating their t-wise coverage or the mutation score, as performed in Part III.

12.3 Preliminary results and future investigations

The collaboration with CETREL is an ongoing work that already led to interesting outcomes. This sections introduces current results and future investigations.



(a) Original set of authorizations

(b) Set covering all the 2-wise interactions

(c) Prioritization of the reduced set

Figure 12.4: Reduction and prioritization of 35,000 authorizations with 2-wise interaction coverage. A set of 35,000 authorizations is reduced to 180 while keeping the same pairwise coverage. Prioritizing these 180 authorizations allows reaching a pairwise coverage greater than 95% with only 10 of them while a random approach needs more than 40 authorizations to reach the same level of coverage.

12.3.1 Current results

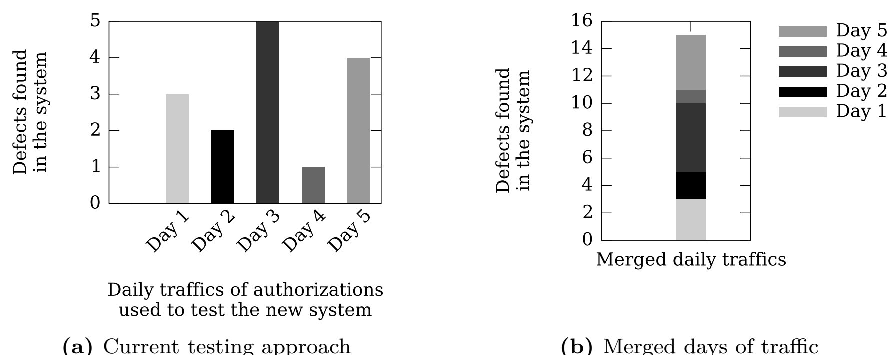
Reduction and prioritization. We applied a 2-wise generation approach as well as a similarity prioritization on a set of 35,000 Authorizations. It allows to reduce to from 35,000 to 180 authorization by keeping the same 2-wise coverage. Regarding the prioritization, the random approach needs more than 4 times the number of authorizations necessary to reach 95% of 2-wise coverage.

Evaluation of the existing test set. CETREL is recurrently using an authorization set of 4,806 authorizations. By analyzing this test set, we found that it covers only 19.67% of the authorization variability. In addition, only a subset of about 40 parameters are used out of 120 possible. It means that the domain covered by this authorization set is only 1/5 of the full authorization domain.

12.3.2 Work in progress and perspectives

Refining the variability model. With the previous results, we found that the test sets used by CETREL always use the same parameters. These parameters are characterized as the most important ones by the experts in CETREL. It means that FM is the theoretical one, which models all the possible authorizations from the protocols specifications, but it does not represent the actual model which is used for testing. We can refine this model by either removing the lest important parameters, putting attributes such as importance to the FM or build a FM using traffic in real time using the reverse-engineering techniques as presented in Part V or machine learning techniques. Such a model will also encompass variability on the values of the different parameters.

Merging daily traffics. The idea behind merging daily traffics is to reach an ultimate authorizations set which will contain all the possible scenarios that will exercise all the parts of the card authorization system. Figure 12.5 depicts an example of what is expected. The redundancy among each day of traffic of Figure 12.5a is removed. The traffics are merged into a big authorization set (Figure 12.5b).



(a) Current testing approach

(b) Merged days of traffic

Figure 12.5: Merging daily traffic of authorizations to optimize the testing process. Several sets of authorizations are merged to form a unique set without redundancy.

Feedback and improvements. This project is still ongoing and requires the constant feedback from the experts of CETREL. In particular, the experts are currently running some authorizations sets that we generated in order to evaluate their ability to find actual defects in the new card authorization systems. Thanks to this constant cooperation, we can adjust the different metrics we propose and focus and specific points they want to address. The method and tools we used are generic and can be extended to add other metrics and strategies.

Benefits for CETREL. The current tool we develop is very fruitful for CETREL. It already allows to prioritize, generate, reduce and evaluate traffic of authorizations. In addition, the tool bridges the different format of authorizations, which was something non-existing for CETREL. Indeed, a daily traffic of authorization is a network message, but what is actually used for testing are XML test cases. There is also a third format for storing authorization into a database. They are now able to convert from one format to the two others, linking the real-world traffic of authorization to the functional testing environment. The current main benefit for CETREL is to be able to evaluate the quality of a daily traffic of authorizations in terms of variability.

12.4 Conclusions

In this chapter, we presented a concrete application of the techniques presented in this dissertation. In a context of credit card authorization which is not a priori related to SPLs, we enabled the way to improve the testing process by modeling the variability of authorizations, thus seeing them as a SPL, and then applying the introduced methods. The project is still ongoing and more results are expected in the following months, but the current outcomes are promising, with an optimization of the test suites used. In particular, the main current outcome for CETREL is to be able to evaluate the quality of daily traffics in terms of variability. Indeed, the variability of a given set of authorizations can be evaluated against the global variability (given by the FM), showing the percentage of coverage for these authorizations.

13

Conclusions and outlook

This chapter concludes the dissertation and presents future research directions.

Contents

13.1 Summary **160**
13.2 Future work and open research questions **160**
13.2.1 Search-based techniques applied to software product lines 160
13.2.2 Combinatorial interaction testing in practice 161

This chapter is organized as follows. Section 13.1 summarizes the contributions of this dissertation before Section 13.2 discusses potential directions for future work.

13.1 Summary

In this work, we presented techniques for enabling testing of SPLs based on FMs using SB approaches combined with constraint solvers.

In the mono-objective configuration generation part, we have introduced a scalable technique based on a similarity heuristic for generating configurations with respect to combinatorial testing. By providing a partial but scalable t-wise approach, the proposed approach outperforms the state of the art tools and allow to scale to large SPLs encompassing more than 6,000 features and with high interaction strengths ($t \geq 6$). We then introduced the first mutation-based approach for generating configurations.

In the third part, we have extended the configuration approaches to support multiple testing objectives. The first chapter of this part introduced a significant improvement over the state of the art to support multiple objectives targeting a single configuration. The following chapter has introduced an improvement to the multi-objective generation technique by supporting objectives target a whole configuration suite. Both these techniques have been evaluated on a large set of SPLs and proven to be effective.

The fourth part was focusing and the evaluation of a given configuration suite. We have first introduced a mutation based approach and linked the mutation criterion to the similarity heuristic introduced in Chapter 4. The following chapter has evaluated the correlation of the mutation criterion with fault detection, demonstrating its ability to be an alternative to CIT.

In the fifth part, the first chapter has introduced an approach for reverse-engineering a SPL and its FM from the source code of SPVs. It is the first fully automated and language-independent technique. Chapter 11 has introduced a search-based technique for automatically evaluating whether reverse-engineered FMs were reflecting the system they model, and when it is not the case, automatically fix them. The approach has been evaluated on Linux and proved its ability to fix actual errors in the constraints of the FM.

Finally, the previous chapter has introduced an industrial application of the proposed configuration and evaluation techniques to the CETREL company. By modeling credit card authorizations as a SPL FM, we enabled the use of the introduced methods, thus reducing the testing effort as long as increasing the level of confidence in the test test suites used by the experts.

13.2 Future work and open research questions

This section describes potential future research directions.

13.2.1 Search-based techniques applied to software product lines

Improving the search process. Future work will investigate alternative ways to improve the search process. Specifically, practices like parameter tuning [AF13], supervised search [JCHP13] or hybrid

approaches involving both constraint-driven and genetic search will be considered. The objectives are to (1) investigate the effects of the parameters on the effectiveness and efficiency of the proposed approach, (2) Find criteria for deciding when to stop the evolution process.

Search-based approaches for SPLs as a service. In this dissertation, we proposed several approaches based on search-based techniques. Although we made the implementations publicly available in most of the cases, it is not that easy for the community or industrial to reuse them. In this context, we plan to include them in framework or application programming interface where the user could set his own fitness functions and testing objectives and select different techniques to apply. This framework will also allow to compare different search techniques for a specific problem.

Beyond Boolean FMs. One limitation of the work presented in this dissertation is that it relies of FMs. These models are Boolean and thus cannot encompass complex and non-Boolean constraints. In future work, we plan to work with SMT solvers in order to handle other type of constraints. In particular, we will analyze whether flattening non-Boolean models to Boolean ones as an impact of the solving process.

13.2.2 Combinatorial interaction testing in practice

CIT in the cloud. Recently, the zcov platform has been released, proposing combinatorial testing for SPLs as a service using the cloud of Amazon and parallel computing. The first performances results are promising, managing to generate the 3-wise interactions for Linux in less than 8 hours, according to the last report [i]. It could be interesting to design or adapt our search-based techniques for these architecture and compare them with CIT techniques.

Interaction faults in practice. There are too few approaches which evaluate the interaction faults in practice. One limitation of the proposed approaches is that they do not evaluate whether they can in practice find interaction faults. Thus, there is no need to evaluate the proposed approaches in practice, and also to compare with other techniques.

A repository with SPL including interaction faults To support the previous future work, we aim at providing a SPL repository containing code of SPLs with actual interaction faults. It requires the manual of interactions faults or the collect of existing ones. Such a repository will be useful to the community for applying their approaches or comparing them with other techniques.

[i] http://zcov.net/2015/01/15/beta-3-wise-performance/

List of papers, tools & services

Papers included in the dissertation:

- 2015
  - Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. Combining multi-objective search and constraint solving for configuring large software product lines. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE 2015)*, 2015
- 2014
  - C Henard, M Papadakis, G Perrouin, J Klein, P Heymans, and Y Le Traon. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Trans. Software Eng.*, 40(7):650–670, 2014
  - Christopher Henard, Mike Papadakis, and Yves Le Traon. Mutation-based generation of software product line test configurations. In Claire Le Goues and Shin Yoo, editors, *Search-Based Software Engineering*, volume 8636 of *Lecture Notes in Computer Science*, pages 92–106. Springer International Publishing, 2014
  - Tewfik Ziadi, Christopher Henard, Mike Papadakis, Mikal Ziane, and Yves Le Traon. Towards a language-independent approach for reverse-engineering of software product lines. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1064–1071, New York, NY, USA, 2014. ACM
  - Mike Papadakis, Christopher Henard, and Yves Le Traon. Sampling program inputs with mutation analysis: Going beyond combinatorial interaction testing. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, pages 1–10, Washington, DC, USA, 2014. IEEE Computer Society
  - Christopher Henard, Mike Papadakis, and Yves Le Traon. Mutalog: A tool for mutating logic formulas. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '14, pages 399–404, Washington, DC, USA, 2014. IEEE Computer Society
- 2013
  - Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Multi-objective test generation for software product lines. In *Proceedings of the 17th International Software Product Line Conference*, pages 62–71. ACM, 2013
  - Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Towards automated testing and fixing of re-engineered feature models. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 1245–1248. IEEE, 2013
  - Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Assessing software product line testing via model-based mutation: An application to similarity testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 188–197. IEEE, 2013
  - Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Pledge: A product line editor and test generation tool. In *Proceedings of the 17th International Software Product Line Conference Co-located Workshops*, SPLC '13 Workshops, pages 126–129, New York, NY, USA, 2013. ACM

Papers not included in the dissertation:

- Antonia Bertolino, Said Daoudagh, Donia El Kateb, Christopher Henard, Yves Le Traon, Francesca Lonetti, Eda Marchetti, Tejeddine Mouelhi, and Mike Papadakis. Similarity testing for access control. *Information and Software Technology*, 58:355–372, 2015
- Christopher Henard, Mike Papadakis, and Yves Le Traon. Flattening or not of the combinatorial interaction testing models? In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, pages 1–4. IEEE, 2015

Papers published prior PhD:

- Thibault Cholez, Christopher Henard, Isabelle Chrisment, Olivier Festor, Guillaume Doyen, and Rida Khatoun. A first approach to detect suspicious peers in the kad p2p network. In *Network and Information Systems Security (SAR-SSI), 2011 Conference on*, pages 1–8. IEEE, 2011

Software developed during PhD:

- *PLEDGE:* A product line editor and configuration generation tool (`http://research.henard.net/SPL/PLEDGE/`)
- *MutaLog:* A tool for mutating logic formulas (`http://research.henard.net/SPL/MutaLog/`)
- *ExtractorPL:* A tool for reverse-engineering a SPL from software product variants (`http://pagesperso-systeme.lip6.fr/Tewfik.Ziadi/sac14/`)

Services:

- Journal reviewer: STVR (in 2014), IST (in 2015)
- Journal co-reviewer: SQJ (in 2014)
- Conference co-reviewer: ISARCS 2012, ICST 2012 & 2013, SAC SVT 2013, ISSRE 2013, ICSE SEIP 2013, MODELS 2013 & 2014, SPLC 2014, ICSE 2014, ICSE NIER 2014.
- Workshop reviewer: JLDP 2012, Mutation 2014
- Webmaster of the ICST 2013 conference website

Bibliography

[AB11] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1–10, New York, NY, USA, 2011. ACM.

[AB12] Andrea Arcuri and Lionel Briand. Formal analysis of the probability of interaction fault detection using random testing. *IEEE Trans. Softw. Eng.*, 38(5):1088–1099, September 2012.

[ABBJ14] Mathieu Acher, Benoit Baudry, Olivier Barais, and Jean-Marc Jézéquel. Customization and 3d printing: A challenging playground for software product lines. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, pages 142–146, New York, NY, USA, 2014. ACM.

[ABHPW10] Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Trans. Softw. Eng.*, 36(6):742–762, November 2010.

[ABL05] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 402–411, New York, NY, USA, 2005. ACM.

[ABLN06] J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *Software Engineering, IEEE Transactions on*, 32(8):608–624, Aug 2006.

[ACC+11] Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien, and Philippe Lahire. Reverse engineering architectural feature models. In *Proceedings of the 5th European Conference on Software Architecture*, ECSA'11, pages 220–235, Berlin, Heidelberg, 2011. Springer-Verlag.

[ACC+14] Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien, and Philippe Lahire. Extraction and evolution of architectural variability models in plugin-based systems. volume 13, pages 1367–1394. Springer Berlin Heidelberg, 2014.

[ACP+12] Mathieu Acher, Anthony Cleve, Gilles Perrouin, Patrick Heymans, Charles Vanbeneden, Philippe Collet, and Philippe Lahire. On extracting feature models from product descriptions. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, pages 45–54, New York, NY, USA, 2012. ACM.

[AF13] Andrea Arcuri and Gordon Fraser. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18(3):594–623, 2013.

[AHH11] Ebrahim Khalil Abbasi, Arnaud Hubaux, and Patrick Heymans. A toolset for feature-based configuration workflows. In *Proceedings of the 2011 15th International Software Product Line Conference*, SPLC '11, pages 65–69, Washington, DC, USA, 2011. IEEE Computer Society.

[AKL09] Sven Apel, Christian Kastner, and Christian Lengauer. Featurehouse: Language-independent, automated software composition. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 221–231, Washington, DC, USA, 2009. IEEE Computer Society.

[ASB+08] Vander Alves, Christa Schwanninger, Luciano Barbosa, Awais Rashid, Peter Sawyer, Paul Rayson, Christoph Pohl, and Andreas Rummler. An exploratory study of information retrieval techniques in domain analysis. In *Proceedings of the 2008 12th International Software Product Line Conference*, SPLC '08, pages 67–76, Washington, DC, USA, 2008. IEEE Computer Society.

[Bat05] Don Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Conference on Software Product Lines*, SPLC'05, pages 7–20, Berlin, Heidelberg, 2005. Springer-Verlag.

[BC07] Renée C. Bryce and Charles J. Colbourn. One-test-at-a-time heuristic search for interaction test suites. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1082–1089, New York, NY, USA, 2007. ACM.

[BDEK+15] Antonia Bertolino, Said Daoudagh, Donia El Kateb, Christopher Henard, Yves Le Traon, Francesca Lonetti, Eda Marchetti, Tejeddine Mouelhi, and Mike Papadakis. Similarity testing for access control. *Information and Software Technology*, 58:355–372, 2015.

[BFN08] Dimo Brockhoff, Tobias Friedrich, and Frank Neumann. Analyzing hypervolume indicator based algorithms. In *Proceedings of the 10th International Conference on Parallel Problem Solving from Nature: PPSN X*, pages 651–660, Berlin, Heidelberg, 2008. Springer-Verlag.

[BM07] Renée C. Bryce and Atif M. Memon. Test suite prioritization by interaction coverage. In *Workshop on Domain Specific Approaches to Software Test Automation: In Conjunction with the 6th ESEC/FSE Joint Meeting*, DOSTA '07, pages 1–7, New York, NY, USA, 2007. ACM.

[BP10] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *JSAT*, 7(2-3):59–6, 2010.

[BRN+13] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A survey of variability modeling in industrial practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 7:1–7:8, New York, NY, USA, 2013. ACM.

[BRR10] Christian Berger, Holger Rendel, and Bernhard Rumpe. Measuring the ability to form a product line from existing products. In *Fourth International Workshop on Variability Modelling of Software-Intensive Systems, Linz, Austria, January 27-29, 2010. Proceedings*, pages 151–154, 2010.

[BSL+10] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. Variability modeling in the real: A perspective from the operating systems domain. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 73–82, New York, NY, USA, 2010. ACM.

[BSL+12] Thorsten Berger, Steven She, Rafael Lotufo, Andrezj Wasowski, and Krzysztof Czarnecki. Variability modeling in the systems software domain. Technical report, Generative Software Development Laboratory, University of Waterloo, 2012.

[BSPM11] RenéeC. Bryce, Sreedevi Sampath, JanB. Pedersen, and Schuyler Manchester. Test suite prioritization by cost-based combinatorial interaction coverage. *International Journal of System Assurance Engineering and Management*, 2(2):126–134, 2011.

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, September 2010.

[BTRC05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering*, CAiSE'05, pages 491–503, Berlin, Heidelberg, 2005. Springer-Verlag.

[BYL+12] Mehra N. Borazjany, Linbin Yu, Yu Lei, Raghu Kacker, and Rick Kuhn. Combinatorial testing of acts: A case study. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 591–600, Washington, DC, USA, 2012. IEEE Computer Society.

[CCL03] Myra B. Cohen, Charles J. Colbourn, and Alan C. H. Ling. Augmenting simulated annealing to build interaction test suites. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE '03, pages 394–, Washington, DC, USA, 2003. IEEE Computer Society.

[CCMJ12] Stephen Creff, Joel Champeau, Arnaud Monegier, and Jean-Marc Jézéquel. Relationships formalization for model-based product lines. In *Proceedings of the 2012 19th Asia-Pacific Software Engineering Conference - Volume 01*, APSEC '12, pages 158–163, Washington, DC, USA, 2012. IEEE Computer Society.

[CDFP97] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The aetg system: An approach to testing based on combinatorial design. *IEEE Trans. Softw. Eng.*, 23(7):437–444, July 1997.

[CDH13] John A. Clark, Haitao Dan, and Robert M. Hierons. Semantic mutation testing. *Sci. Comput. Program.*, 78(4):345–363, April 2013.

[CDS06] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Coverage and adequacy in software product line testing. In *Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis*, ROSATEA '06, pages 53–63, New York, NY, USA, 2006. ACM.

[CDS07] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 129–139, New York, NY, USA, 2007. ACM.

[CDS08] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Trans. Softw. Eng.*, 34(5):633–650, September 2008.

[CG08] Andrea Calvagna and Angelo Gargantini. A logic-based approach to combinatorial testing with constraints. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 66–83, Berlin, Heidelberg, 2008. Springer-Verlag.

[CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, July 2006.

[CHC+11] Thibault Cholez, Christopher Henard, Isabelle Chrisment, Olivier Festor, Guillaume Doyen, and Rida Khatoun. A first approach to detect suspicious peers in the kad p2p network. In *Network and Information Systems Security (SAR-SSI), 2011 Conference on*, pages 1–8. IEEE, 2011.

[CHS08] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. What's in a feature: A requirements engineering perspective. In *Proceedings of the Theory and Practice of Software, 11th International Conference on Fundamental Approaches to Software Engineering*, FASE'08/ETAPS'08, pages 16–30, Berlin, Heidelberg, 2008. Springer-Verlag.

[CHW98] J. Coplien, D. Hoffman, and D. Weiss. Commonality and variability in software engineering. *Software, IEEE*, 15(6):37–45, Nov 1998.

[Cli96] Norman Cliff. *Ordinal Methods for Behavioral Data Analysis.* Psychology Press, New-York, USA, September 1996.

[CN01] Paul C. Clements and Linda Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[DAVV98] Gary B. Lamont David A. Van Veldhuizen. Multiobjective evolutionary algorithm research: A history and analysis. Technical report, tr-98-03, Department of Electrical and Computer Engineering, Air Force Institute of Technology, 1998.

[DDH+13] Jean-Marc Davril, Edouard Delfosse, Negar Hariri, Mathieu Acher, Jane Cleland-Huang, and Patrick Heymans. Feature model extraction from large collections of informal product descriptions. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 290–300, New York, NY, USA, 2013. ACM.

[DER05] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, October 2005.

[DJW02] Stefan Droste, Thomas Jansen, and Ingo Wegener. On the analysis of the (1+ 1) evolutionary algorithm. *Theor. Comput. Sci.*, 276(1-2):51–81, April 2002.

[DMB11] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011.

[DMSNCMM+11] Paulo Anselmo Da Mota Silveira Neto, Ivan Do Carmo Machado, John D. Mcgregor, Eduardo Santana De Almeida, and Silvio Romero De Lemos Meira. A systematic mapping study of software product lines testing. *Inf. Softw. Technol.*, 53(5):407–423, May 2011.

[DN11] Juan J. Durillo and Antonio J. Nebro. jmetal: A java framework for multi-objective optimization. *Adv. Eng. Softw.*, 42(10):760–771, October 2011.

[DO91] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17(9):900–910, September 1991.

[DPAM02] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Trans. Evol. Comp*, 6(2):182–197, April 2002.

[DR06] Hyunsook Do and Gregg Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Trans. Softw. Eng.*, 32(9):733–752, September 2006.

[EBG12] Faezeh Ensan, Ebrahim Bagheri, and Dragan Gašević. Evolutionary search-based test generation for software product line feature models. In *Proceedings of the 24th International Conference on Advanced Information Systems Engineering*, CAiSE'12, pages 613–628, Berlin, Heidelberg, 2012. Springer-Verlag.

[ER11] Emelie Engström and Per Runeson. Software product line testing - a systematic mapping study. *Inf. Softw. Technol.*, 53(1):2–13, January 2011.

[FHLS98] P.G. Frankl, R.G. Hamlet, Bev Littlewood, and L. Strigini. Evaluating testing methods by delivered reliability [software]. *Software Engineering, IEEE Transactions on*, 24(8):586–601, Aug 1998.

[FKBA07] P. Frenzel, R. Koschke, A.P.J. Breu, and K. Angstmann. Extending the reflexion method for consolidating software variants into product lines. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 160–169, Oct 2007.

[FMM+96] Sandra Camargo Pinto Ferraz Fabbri, José C. Maldonado, Paulo Cesar Masiero, Márcio E. Delamaro, and E. Wong. Mutation testing applied to validate specifications based on petri nets. In *Proceedings of the IFIP TC6 Eighth International Conference on Formal Description Techniques VIII*, pages 329–337, London, UK, UK, 1996. Chapman & Hall, Ltd.

[FPDN05] Alan M. Frisch, Timothy J. Peugniez, Anthony J. Doggett, and Peter W. Nightingale. Solving non-boolean satisfiability problems with stochastic local search: A comparison of encodings. volume 35, pages 143–179. Springer-Verlag New York, Inc., Secaucus, NJ, USA, October 2005.

[FZ11] Gordon Fraser and Andreas Zeller. Generating parameterized unit tests. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 364–374, New York, NY, USA, 2011. ACM.

[FZ12] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *Software Engineering, IEEE Transactions on*, 38(2):278–292, March 2012.

[GCD11] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Softw. Engg.*, 16(1):61–102, February 2011.

[GF11] Angelo Gargantini and Gordon Fraser. Generating minimal fault detecting test suites for general boolean specifications. *Information & Software Technology*, 53(11):1263–1273, 2011.

[GGZ+13] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 302–313, New York, NY, USA, 2013. ACM.

[GLOA06] Mats Grindal, Birgitta Lindström, Jeff Offutt, and Sten F. Andler. An evaluation of combination strategies for test case selection. *Empirical Softw. Engg.*, 11(4):583–611, December 2006.

[GOA05] Mats Grindal, Jeff Offutt, and Sten F. Andler. Combination testing strategies: a survey. *Softw. Test., Verif. Reliab.*, 15(3):167–199, 2005.

[GWW+11] Jianmei Guo, Jules White, Guangxin Wang, Jian Li, and Yinglin Wang. A genetic algorithm for optimized feature selection with resource constraints in software product lines. *J. Syst. Softw.*, 84(12):2208–2221, December 2011.

[Haw01] Thomas Hawkins. *Lebesgue's theory of integration: its origins and development*, volume 282. American Mathematical Soc., 2001.

[HB10] Hadi Hemmati and Lionel Briand. An industrial investigation of similarity measures for model-based test case selection. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*, ISSRE '10, pages 141–150, Washington, DC, USA, 2010. IEEE Computer Society.

[HBG11] Aymeric Hervieu, Benoit Baudry, and Arnaud Gotlieb. Pacogen: Automatic generation of pairwise test configurations from feature models. In *Proceedings of the 2011 IEEE 22Nd International Symposium on Software Reliability Engineering*, ISSRE '11, pages 120–129, Washington, DC, USA, 2011. IEEE Computer Society.

[HJL11] Mark Harman, Yue Jia, and William B. Langdon. Strong higher order mutation-based test data generation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 212–222, New York, NY, USA, 2011. ACM.

[HM10] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Trans. Softw. Eng.*, 36(2):226–247, March 2010.

[HMZ12] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, 45(1):11:1–11:61, December 2012.

[HPHLT15] Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. Combining multi-objective search and constraint solving for configuring large software product lines. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE 2015)*, 2015.

[HPLT14] Christopher Henard, Mike Papadakis, and Yves Le Traon. Mutation-based generation of software product line test configurations. In Claire Le Goues and Shin Yoo, editors, *Search-Based Software Engineering*, volume 8636 of *Lecture Notes in Computer Science*, pages 92–106. Springer International Publishing, 2014.

[HPLT15] Christopher Henard, Mike Papadakis, and Yves Le Traon. Flattening or not of the combinatorial interaction testing models? In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, pages 1–4. IEEE, 2015.

[HPP+12] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test suites for large software product lines. Technical report, 2012.

[HPP+13a] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Assessing software product line testing via model-based mutation: An application to similarity testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 188–197. IEEE, 2013.

[HPP+13b] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Towards automated testing and fixing of re-engineered feature models. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 1245–1248. IEEE, 2013.

[HPP+13c] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Multi-objective test generation for software product lines. In *Proceedings of the 17th International Software Product Line Conference*, pages 62–71. ACM, 2013.

[HPP+13d] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Pledge: A product line editor and test generation tool. In *Proceedings of the 17th International Software Product Line Conference Co-located Workshops*, SPLC '13 Workshops, pages 126–129, New York, NY, USA, 2013. ACM.

[HPP+14] C Henard, M Papadakis, G Perrouin, J Klein, P Heymans, and Y Le Traon. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Trans. Software Eng.*, 40(7):650–670, 2014.

[HPT14] Christopher Henard, Mike Papadakis, and Yves Le Traon. Mutalog: A tool for mutating logic formulas. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '14, pages 399–404, Washington, DC, USA, 2014. IEEE Computer Society.

[Jac01] Paul Jaccard. Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bulletin del la Société Vaudoise des Sciences Naturelles*, 37:547–579, 1901.

[JCHP13] Yue Jia, Myra B Cohen, Mark Harman, and Justyna Petke. Learning combinatorial interaction testing strategies using hyperheuristic search. *RN/UCL*, 13:17, 2013.

[JH09] Yue Jia and Mark Harman. Higher order mutation testing. *Inf. Softw. Technol.*, 51(10):1379–1393, October 2009.

[JH11] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.*, 37(5):649–678, September 2011.

[JHF11] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. Properties of realistic feature models make combinatorial testing of product lines feasible. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, MODELS'11, pages 638–652, Berlin, Heidelberg, 2011. Springer-Verlag.

[JHF12a] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, pages 46–55, New York, NY, USA, 2012. ACM.

[JHF+12b] Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Anne Grete Eldegard, and Torbjørn Syversen. Generating better partial covering arrays by modeling weights on sub-product lines. In *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems*, MODELS'12, pages 269–284, Berlin, Heidelberg, 2012. Springer-Verlag.

[JHF+12c] MartinFagereng Johansen, Øystein Haugen, Franck Fleurey, Erik Carlson, Jan Endresen, and Tormod Wien. A technique for agile and automatic interaction testing for product lines. In Brian Nielsen and Carsten Weise, editors, *Testing Software and Systems*, volume 7641 of *Lecture Notes in Computer Science*, pages 39–54. Springer Berlin Heidelberg, 2012.

[JJI+14] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 654–665, New York, NY, USA, 2014. ACM.

[KAO13] Gary Kaminski, Paul Ammann, and Jeff Offutt. Improving logic-based testing. *J. Syst. Softw.*, 86(8):2002–2012, August 2013.

[KBK11] Chang Hwan Peter Kim, Don S. Batory, and Sarfraz Khurshid. Reducing combinatorics in testing product lines. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 57–68, New York, NY, USA, 2011. ACM.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-21, 1990.

[KCS06] Abdullah Konak, David W. Coit, and Alice E. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering & System Safety*, 91(9):992–1007, September 2006.

[Ken38] M. G. Kendall. A New Measure of Rank Correlation. *Biometrika*, 30(1/2):81–93, June 1938.

[KK12] Benjamin Klatt and Martin Küster. Respecting component architecture to migrate product copies to a software product line. In *Proceedings of the 17th International Doctoral Symposium on Components and Architecture*, WCOP '12, pages 7–12, New York, NY, USA, 2012. ACM.

[KLD02] K.C. Kang, Jaejoon Lee, and P. Donohoe. Feature-oriented product line engineering. *Software, IEEE*, 19(4):58–65, Jul 2002.

[KLK08] Rick Kuhn, Yu Lei, and Raghu Kacker. Practical combinatorial testing: Beyond pairwise. *IT Professional*, 10(3):19–23, May 2008.

[KPAO11] Garrett Kent Kaminski, Upsorn Praphamontripong, Paul Ammann, and Jeff Offutt. A logic mutation approach to selective mutation for programs and queries. *Information & Software Technology*, 53(10):1137–1152, 2011.

[Kru02] Charles W. Krueger. Easing the transition to software mass customization. In *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, PFE '01, pages 282–293, London, UK, UK, 2002. Springer-Verlag.

[KSP09] Kyo C. Kang, Vijayan Sugumaran, and Sooyong Park. *Applied Software Product Line Engineering*. Auerbach Publications, Boston, MA, USA, 1st edition, 2009.

[KTS+09] Christian Kastner, Thomas Thum, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. Featureide: A tool framework for feature-oriented software development. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 611–614, Washington, DC, USA, 2009. IEEE Computer Society.

[KTZ06] J. Knowles, L. Thiele, and E. Zitzler. A Tutorial on the Performance Assessment of Stochastic Multiobjective Optimizers. TIK Report 214, Computer Engineering and Networks Laboratory (TIK), ETH Zurich, February 2006.

[KWG04] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, Jr. Software fault interactions and implications for software testing. *IEEE Trans. Softw. Eng.*, 30(6):418–421, June 2004.

[LBP10] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation(JSAT)*, 7:59–64, 2010.

[LGDVFW12] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 3–13, Piscataway, NJ, USA, 2012. IEEE Press.

[LH12] William B. Langdon and Mark Harman. Genetically improving 50000 lines of C++. Research Note RN/12/09, Department of Computer Science, University College London, 19 September 2012.

[LHB01] RobertoE. Lopez-Herrejon and Don Batory. A standard problem for evaluating product-line methodologies. In Jan Bosch, editor, *Generative and Component-Based Software Engineering*, volume 2186 of *Lecture Notes in Computer Science*, pages 10–24. Springer Berlin Heidelberg, 2001.

[LHGB+12] Roberto Erick Lopez-Herrejon, José A. Galindo, David Benavides, Sergio Segura, and Alexander Egyed. Reverse engineering feature models with evolutionary algorithms: An exploratory study. In *Proceedings of the 4th International Conference on Search Based Software Engineering*, SSBSE'12, pages 168–182, Berlin, Heidelberg, 2012. Springer-Verlag.

[lin] Tools for analyzing variability in the linux kernel. `http://code.google.com/p/linux-variability-analysis-tools/source/browse/?repo=formulas`.

[LLWG12] Jian Li, Xijuan Liu, Yinglin Wang, and Jianmei Guo. Formalizing feature selection problem in software product lines using 0-1 programming. In Yinglin Wang and Tianrui Li, editors, *Practical Applications of Intelligent Systems*, volume 124 of *Advances in Intelligent and Soft Computing*, pages 459–465. Springer Berlin Heidelberg, 2012.

[LP07] Felix Loesch and Erhard Ploedereder. Optimization of variability in software product lines. In *Proceedings of the 11th International Software Product Line Conference*, SPLC '07, pages 151–162, Washington, DC, USA, 2007. IEEE Computer Society.

[LT98] Yu Lei and Kuo-Chung Tai. In-parameter-order: A test generation strategy for pairwise testing. In *The 3rd IEEE International Symposium on High-Assurance Systems Engineering*, HASE '98, pages 254–261, Washington, DC, USA, 1998. IEEE Computer Society.

[LY14] Per Kristian Lehre and Xin Yao. Runtime analysis of the (1 + 1) ea on computing unique input output sequences. *Inf. Sci.*, 259:510–531, 2014.

[MA04] R.T. Marler and J.S. Arora. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, 26(6):369–395, 2004.

[MBC09] Marcilio Mendonca, Moises Branco, and Donald Cowan. S.p.l.o.t.: Software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 761–762, New York, NY, USA, 2009. ACM.

[MBLT06] Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. Mutation analysis testing for model transformations. In *Proceedings of the Second European Conference on Model Driven Architecture: Foundations and Applications*, ECMDA-FA'06, pages 376–390, Berlin, Heidelberg, 2006. Springer-Verlag.

[McG10] JohnD. McGregor. Testing a software product line. In Paulo Borba, Ana Cavalcanti, Augusto Sampaio, and Jim Woodcook, editors, *Testing Techniques in Software Engineering*, volume 6153 of *Lecture Notes in Computer Science*, pages 104–140. Springer Berlin Heidelberg, 2010.

[MFB08] T. Mouelhi, F. Fleurey, and B. Baudry. A generic metamodel for security policies mutation. In *Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conference on*, pages 278 –286, april 2008.

[MI07] John D. McGregor and Kyungsoo Im. The implications of variation for testing in a software product line. pages 59–64, 2007.

[MWC09] Marcilio Mendonca, Andrzej Wąsowski, and Krzysztof Czarnecki. Sat-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 231–240, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[NFLTJ04] Clémentine Nebut, Franck Fleurey, Yves Le Traon, and Jean-Marc Jézéquel. A requirement-based approach to test product families. In FrankJ. van der Linden, editor, *Software Product-Family Engineering*, volume 3014 of *Lecture Notes in Computer Science*, pages 198–210. Springer Berlin Heidelberg, 2004.

[NKN14] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 907–918, New York, NY, USA, 2014. ACM.

[NL11] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2):11:1–11:29, February 2011.

[NPLTJ03] C. Nebut, S. Pickin, Y. Le Traon, and J. Jézéquel. Automated requirements-based generation of test cases for product families. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 263–266, Oct 2003.

[NTJ06] Clémentine Nebut, YvesLe Traon, and Jean-Marc Jézéquel. System testing of product lines: From requirements to test cases. In Timo Käköla and JuanCarlos Duenas, editors, *Software Product Lines*, pages 447–477. Springer Berlin Heidelberg, 2006.

[Off11] Jeff Offutt. A mutation carol: Past, present and future. *Information & Software Technology*, 53(10):1098–1107, 2011.

[Ola13] Rafael Olaechea. Optimization of variability in software product lines. Master's thesis, University of Waterloo, Ontario, 2013.

[OMR10] Sebastian Oster, Florian Markert, and Philipp Ritter. Automated incremental pairwise testing of software product lines. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*, SPLC'10, pages 196–210, Berlin, Heidelberg, 2010. Springer-Verlag.

[ORGC14] Rafael Olaechea, Derek Rayside, Jianmei Guo, and Krzysztof Czarnecki. Comparison of exact and approximate multi-objective optimization for software product lines. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, pages 92–101, New York, NY, USA, 2014. ACM.

[OSCR12] Rafael Olaechea, Steven Stewart, Krzysztof Czarnecki, and Derek Rayside. Modelling and multi-objective optimization of quality attributes in variability-rich software. In *Proceedings of the Fourth International Workshop on Nonfunctional System Properties in Domain Specific Modeling Languages*, NFPinDSML '12, pages 2:1–2:6, New York, NY, USA, 2012. ACM.

[oST] British Computer Society. Special Interest Group on Software Testing. *Standard for Software Component Testing.*

[PBL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[Pet15] Justyna Petke. Constraints: the future of combinatorial interaction testing. In *Proceedings of the 8th International Workshop on Search-Based Software Testing*, SBST '15, 2015. To appear.

[PHT14] Mike Papadakis, Christopher Henard, and Yves Le Traon. Sampling program inputs with mutation analysis: Going beyond combinatorial interaction testing. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, pages 1–10, Washington, DC, USA, 2014. IEEE Computer Society.

[PLP11] R. Pohl, K. Lauenroth, and K. Pohl. A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 313–322, Nov 2011.

[PLT12] M. Papadakis and Y. Le Traon. Using mutants to locate "unknown" faults. pages 691–700, April 2012.

[PLT13] Mike Papadakis and Yves Le Traon. Metallaxis-fl: mutation-based fault localization. *Software Testing, Verification and Reliability*, pages n/a–n/a, 2013.

[PM11] Mike Papadakis and Nicos Malevris. Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. *Software Quality Control*, 19(4):691–723, December 2011.

[PM12] Mike Papadakis and Nicos Malevris. Mutation based test case generation via a path selection strategy. *Inf. Softw. Technol.*, 54(9):915–932, September 2012.

[POS+12] Gilles Perrouin, Sebastian Oster, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Traon. Pairwise testing for software product lines: Comparison of two approaches. *Software Quality Control*, 20(3-4):605–643, September 2012.

[PSK+10] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 459–468, Washington, DC, USA, 2010. IEEE Computer Society.

[PYCH13] Justyna Petke, Shin Yoo, Myra B. Cohen, and Mark Harman. Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 26–36, New York, NY, USA, 2013. ACM.

[QCR08] Xiao Qu, Myra B. Cohen, and Gregg Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, pages 75–86, 2008.

[RC12] Julia Rubin and Marsha Chechik. Combining related products into product lines. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering*, FASE'12, pages 285–300, Berlin, Heidelberg, 2012. Springer-Verlag.

[RE12] Per Runeson and Emelie Engstrom. Software product line testing – a 3d regression testing problem. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 742–746, Washington, DC, USA, 2012. IEEE Computer Society.

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education, 2 edition, 2003.

[ROZ97] Dan Rubenstein, Leon Osterweil, and Shlomo Zilberstein. An anytime approach to analyzing software systems. In *Proceedings of the 10th International Florida Artificial Intelligence Research Symposium*, pages 386–391, 1997.

[RPK10] Uwe Ryssel, Joern Ploennigs, and Klaus Kabitzsch. Automatic variation-point identification in function-block-based models. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 23–32, New York, NY, USA, 2010. ACM.

[RSM+10] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 445–454, New York, NY, USA, 2010. ACM.

[SBV+08] Sreedevi Sampath, Renee C. Bryce, Gokulanand Viswanath, Vani Kandimalla, and A. Gunes Koru. Prioritizing user-session-based test cases for web applications testing. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, ICST '08, pages 141–150, Washington, DC, USA, 2008. IEEE Computer Society.

[SGB+12] Sergio Segura, José A. Galindo, David Benavides, José A. Parejo, and Antonio Ruiz-Cortés. Betty: Benchmarking and testing on the automated analysis of feature models. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, pages 63–71, New York, NY, USA, 2012. ACM.

[SGPMA13] Abdel Salam Sayyad, Katerina Goseva-Popstojanova, Tim Menzies, and Hany Ammar. On parameter tuning in search based software engineering: A replicated empirical study. In *Proceedings of the 2013 3rd International Workshop on Replication in Empirical Software Engineering Research*, RESER '13, pages 84–90, Washington, DC, USA, 2013. IEEE Computer Society.

[SHT06] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature diagrams: A survey and a formal semantics. In *Proceedings of the 14th IEEE International Requirements Engineering Conference*, RE '06, pages 136–145, Washington, DC, USA, 2006. IEEE Computer Society.

[SIMA13a] Abdel Salam Sayyad, Joseph Ingram, Tim Menzies, and Hany Ammar. Optimum feature selection in software product lines: Let your model and values guide your search. In *Proceedings of the 1st International Workshop on Combining Modelling and Search-Based Software Engineering*, CMSBSE '13, pages 22–27, Piscataway, NJ, USA, 2013. IEEE Press.

[SIMA13b] A.S. Sayyad, J. Ingram, T. Menzies, and H. Ammar. Scalable product line configuration: A straw to break the camel's back. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 465–474, Nov 2013.

[SLB+11] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 461–470, New York, NY, USA, 2011. ACM.

[SMA13] Abdel Salam Sayyad, Tim Menzies, and Hany Ammar. On the value of user preferences in search-based software engineering: A case study in software product lines. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 492–501, Piscataway, NJ, USA, 2013. IEEE Press.

[SNX05] Liang Shi, Changhai Nie, and Baowen Xu. A software debugging method based on pairwise testing. In *Proceedings of the 5th International Conference on Computational Science - Volume Part III*, ICCS'05, pages 1088–1091, Berlin, Heidelberg, 2005. Springer-Verlag.

[SOLF12] Michaela Steffens, Sebastian Oster, Malte Lochau, and Thomas Fogdal. Industrial evaluation of pairwise spl testing with moso-polite. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, pages 55–62, New York, NY, USA, 2012. ACM.

[SPF12] Charles Song, Adam Porter, and Jeffrey S. Foster. itree: Efficiently discovering high-coverage configurations using interaction trees. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 903–913, Piscataway, NJ, USA, 2012. IEEE Press.

[TBK09] Thomas Thum, Don Batory, and Christian Kastner. Reasoning about edits to feature models. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 254–264, Washington, DC, USA, 2009. IEEE Computer Society.

[TKB+14] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. Featureide: An extensible framework for feature-oriented software development. *Sci. Comput. Program.*, 79:70–85, January 2014.

[TSD+12] Reinhard Tartler, Julio Sincero, Christian Dietrich, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Revealing and repairing configuration inconsistencies in large-scale system software. *International Journal on Software Tools for Technology Transfer*, 14(5):531–551, 2012.

[UKB10] Engin Uzuncaova, Sarfraz Khurshid, and Don Batory. Incremental test generation for software product lines. volume 36, pages 309–322, Piscataway, NJ, USA, May 2010. IEEE Press.

[VBP12] M.T. Valente, V. Borges, and L. Passos. A semi-automatic approach for extracting software product lines. *Software Engineering, IEEE Transactions on*, 38(4):737–754, July 2012.

[VD00] A. Vargha and H. D. Delaney. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal on Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[WDS08] Jules White, Brian Doughtery, and Douglas C. Schmidt. Filtered cartesian flattening: An approximation technique for optimally selecting features while adhering to resource constraints. In *12th International Conference on Software Product Lines*, pages 209–216, September 2008.

[WDS09] Jules White, Brian Dougherty, and Douglas C. Schmidt. Selecting highly optimal architectural feature sets with filtered cartesian flattening. *J. Syst. Softw.*, 82(8):1268–1284, August 2009.

[WGS+14] Jules White, José A. Galindo, Tripti Saxena, Brian Dougherty, David Benavides, and Douglas C. Schmidt. Evolving feature model configurations in software product lines. *J. Syst. Softw.*, 87:119–136, January 2014.

[WRH+00] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[WTKC11] Zhiqiao Wu, Jiafu Tang, C. K. Kwong, and Ching-Yuen Chan. An optimization model for reuse scenario selection considering reliability and cost in software product line development. *International Journal of Information Technology and Decision Making*, 10(5):811–841, 2011.

[XCMR13] Zhihong Xu, Myra B. Cohen, Wayne Motycka, and Gregg Rothermel. Continuous test suite augmentation in software product lines. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 52–61, New York, NY, USA, 2013. ACM.

[XHSC12] Yingfei Xiong, Arnaud Hubaux, Steven She, and Krzysztof Czarnecki. Generating range fixes for software configuration. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 58–68, Piscataway, NJ, USA, 2012. IEEE Press.

[XXJ12] Yinxing Xue, Zhenchang Xing, and Stan Jarzabek. Feature location in a collection of product variants. In *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*, pages 145–154, 2012.

[YCP06] C. Yilmaz, M.B. Cohen, and A.A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *Software Engineering, IEEE Transactions on*, 32(1):20–34, Jan 2006.

[YH12] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, March 2012.

[YHC13] Shin Yoo, Mark Harman, and David Clark. Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. *ACM Trans. Softw. Eng. Methodol.*, 22(3):19:1–19:29, July 2013.

[YHTS09] Shin Yoo, Mark Harman, Paolo Tonella, and Angelo Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 201–212, New York, NY, USA, 2009. ACM.

[YNHK08] Kentaro Yoshimura, Fumio Narisawa, Koji Hashimoto, and Tohru Kikuno. Fave: Factor analysis based approach for detecting product line variability from change history. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 11–18, New York, NY, USA, 2008. ACM.

[ZB13] Bo Zhang and Martin Becker. Recovar: A solution framework towards reverse engineering variability. In *4th International Workshop on Product LinE Approaches in Software Engineering, PLEASE 2013, San Francisco, CA, USA, May 20, 2013*, pages 45–48, 2013.

[ZBT07] Eckart Zitzler, Dimo Brockhoff, and Lothar Thiele. The hypervolume indicator revisited: On the design of pareto-compliant indicators via weighted integration. In *Proceedings of the 4th International Conference on Evolutionary Multi-criterion Optimization*, EMO'07, pages 862–876, Berlin, Heidelberg, 2007. Springer-Verlag.

[ZFdSZ12] Tewfik Ziadi, Luz Frias, Marcos Aurelio Almeida da Silva, and Mikal Ziane. Feature identification from the source code of product variants. In *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*, CSMR '12, pages 417–422, Washington, DC, USA, 2012. IEEE Computer Society.

[ZHJ04] Tewfik Ziadi, Loïc Hélouët, and Jean-Marc Jézéquel. Towards a uml profile for software product lines. In FrankJ. van der Linden, editor, *Software Product-Family Engineering*, volume 3014 of *Lecture Notes in Computer Science*, pages 129–139. Springer Berlin Heidelberg, 2004.

[ZHP+14] Tewfik Ziadi, Christopher Henard, Mike Papadakis, Mikal Ziane, and Yves Le Traon. Towards a language-independent approach for reverse-engineering of software product lines. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1064–1071, New York, NY, USA, 2014. ACM.

[ZJ06] Tewfik Ziadi and Jean-Marc Jézéquel. Software product line engineering with the UML: deriving products. In *Software Product Lines - Research Issues in Engineering and Management*, pages 557–588. 2006.

[ZK04] Eckart Zitzler and Simon Künzli. Indicator-based selection in multiobjective search. In Xin Yao, EdmundK. Burke, JoséA. Lozano, Jim Smith, JuanJulián Merelo-Guervós, JohnA. Bullinaria, JonathanE. Rowe, Peter Tiňo, Ata Kabán, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VIII*, volume 3242 of *Lecture Notes in Computer Science*, pages 832–842. Springer Berlin Heidelberg, 2004.

[ZTL+03] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and V. G. da Fonseca. Performance assessment of multiobjective optimizers: An analysis and review. *Trans. Evol. Comp*, 7(2):117–132, April 2003.

[ZYL11] Guoheng Zhang, Huilin Ye, and Yuqing Lin. Using knowledge-based systems to manage quality attributes in software product lines. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC '11, pages 32:1–32:7, New York, NY, USA, 2011. ACM.