

**AVALIAÇÃO DE ESTRATÉGIAS DE TESTES
PARA SISTEMAS DE SOFTWARE
CONFIGURÁVEIS**

FISCHER JÔNATAS FERREIRA

**AVALIAÇÃO DE ESTRATÉGIAS DE TESTES
PARA SISTEMAS DE SOFTWARE
CONFIGURÁVEIS**

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

ORIENTADOR: EDUARDO FIGUEIREDO

Belo Horizonte

Agosto de 2021

FISCHER JÔNATAS FERREIRA

**EVALUATING TESTING STRATEGIES FOR
CONFIGURABLE SOFTWARE SYSTEMS**

Thesis presented to the Graduate Program
in Computer Science of the Federal Univer-
sity of Minas Gerais in partial fulfillment of
the requirements for the degree of Doctor
in Computer Science.

ADVISOR: EDUARDO FIGUEIREDO

Belo Horizonte

August 2021

Ferreira, Fischer Jônatas.

F383a Avaliação de estratégias de testes para sistemas de software configuráveis [manuscrito] / Fischer Jônatas Ferreira, 2021. xiv, 138 f. il.

Orientador: Eduardo Magno Lajes Figueiredo.
Tese (Doutorado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f.118-138.

1. Computação – Teses. 2. Software – Ferramentas – Testes – Teses. 4. Sistemas configuráveis – Testes – Teses. I. Figueiredo, Eduardo Magno Lajes II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. III. Título.

CDU 519.6*32(043)




UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Avaliação de Estratégias de Testes para Sistemas Configuráveis de Software

FISCHER JÔNATAS FERREIRA

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:


PROF. EDUARDO MAGNO LAGES FIGUEIREDO - Orientador
Departamento de Ciência da Computação - UFMG


PROF. MARCO TULLIO DE OLIVEIRA VALENTE
Departamento de Ciência da Computação - UFMG


PROF. ANDRÉ CAVALCANTE HORA
Departamento de Ciência da Computação - UFMG


PROF. VANDER RAMOS ALVES
Departamento de Ciência da Computação - UnB


PROF. IVAN DO CARMO MACHADO
Departamento de Ciência da Computação - UFBA

Belo Horizonte, 13 de Agosto de 2021.

Acknowledgments

First of all, All Glory and Praises to Jesus Christ!

This work would not have been possible without the support of many people. They made many contributions to the execution of this research and the fulfillment of the requirements of the Ph.D. program.

I could not have completed this course without the continued support of my beloved wife, Michele. There were many moments of understanding, patience, and support. Indeed, without her, I would not have even started this journey. Therefore, I dedicate my work to my eternal girlfriend with whom I share this achievement.

I thank my father, Messias, and my mother, Angela, for their all love, support, and encouragement dedicated throughout my life. I would also like to devote a special acknowledgment to my advisor, Eduardo Figueiredo. Thanks for all the advice, help, understanding, and patience. Thank you for sharing all the moments of this research and always serving me with great patience, dedication, and support.

I thank all my friends at LabSoft who have contributed and walked with me all these years. I am also grateful for the precise and comforting guidelines of my friends Julio Reis and Maurício Souza. The companions of friends Bruno Luan and Racyus Delano. Also highlighted is the support of friends Johnatan Alves, Cleiton Tavares, and Kattiana Constantino. I especially thank Gustavo Vale and João Paulo Diniz for their contribution to several works that make up this thesis.

I want to express my gratitude to the member of my thesis defense committee: Vander Alves (UnB), Ivan Machado (UFBA), Marco Túlio (UFMG), and Andre Hora (UFMG). I thank the Department of Computer Science at UFMG for the opportunity to participate in its Ph.D. program and provide a course with a high level of excellence. In addition, I thank CAPES for the financial support provided throughout these three years of the Ph.D. course.

Finally, I thank everyone who has contributed directly or indirectly to the conduct of this work and for having encouraged me always to take it forward.

*“O sábio ouvirá e crescerá em conhecimento, e o entendido adquirirá sábios
conselhos.”*

(Provérbios 1:5)

Resumo

Sistemas de software configuráveis permitem que desenvolvedores mantenham uma plataforma única atendendo a uma diversidade de contextos, usos e implantações. Os testes de sistemas configuráveis são essenciais porque as configurações que falham podem prejudicar os usuários e degradar a reputação do projeto. No entanto, testar sistemas configuráveis é muito desafiador devido ao número de configurações a serem executadas em cada teste, levando a uma explosão combinatória do número de configurações e testes. Atualmente, várias estratégias de teste foram propostas para lidar com esse desafio, mas suas potenciais aplicações práticas permanecem amplamente inexploradas. Na verdade, as comparações preliminares existentes de estratégias de testes não visam um conjunto uniforme de sistemas configuráveis. Com base em um grande conjunto de dados de 30 sistemas configuráveis, esta tese compara várias estratégias para testar sistemas configuráveis encontrados por um estudo de mapeamento sistemático. No primeiro estudo, foi projetado e realizado um estudo empírico comparativo com as duas principais ferramentas de teste sólido, chamadas VarexJ e SPLat. Em um segundo estudo empírico foram comparadas dezesseis estratégias de testes pareados. Com a experiência adquirida por meio dos estudos empíricos foi proposta uma lista de dez desafios enfrentados ao criar as suítes de teste para sistemas configuráveis. Ainda, foi relatado como os autores lidaram com as suítes de teste para o conjunto de sistemas descritos nesta tese. A lista proposta inclui, por exemplo, os desafios de testar classes de alto acoplamento e de determinar métricas para medir a qualidade do conjunto de testes. Os resultados dos estudos empíricos indicam quais e quando as estratégias de testes são mais rápidas e eficazes para identificar falhas em sistemas configuráveis. No geral, os autores acreditam que os profissionais podem adquirir o conhecimento necessário por meio dos resultados alcançados, a fim de escolherem uma estratégia de teste que melhor se adapte às suas necessidades e ainda, os profissionais podem se beneficiar com as soluções propostas para cada desafio.

Palavras-chave: Teste de Sistemas Configuráveis, Ferramentas de Teste para Sistemas Configuráveis, Falhas de Interação de Features.

Abstract

Configurable software systems allow developers to maintain a unique platform and address a diversity of deployment contexts and usages. Testing configurable systems is essential because configurations that fail may potentially hurt users and degrade the project reputation. However, testing configurable systems is challenging due to the number of configurations to run with each test suite, leading to a combinatorial explosion in the number of configurations and tests. Currently, several testing strategies have been proposed to deal with this challenge, but their potential practical application remains largely unexplored. In fact, existing comparisons of testing strategies do not rely on a uniform dataset of configurable software systems. Based on a large dataset of 30 configurable systems, this thesis compares several strategies for testing configurable systems found by a systematic mapping study on two empirical studies. In the first study, we designed and performed a comparative empirical study of the two main sound testing tools, namely VarexJ and SPLat. We also compare sixteen t-wise testing strategies in a second empirical study. With the experience we have in the empirical studies, we propose a list of ten challenges faced when creating test suites for configurable systems and dealing with a test suite for our dataset systems. Our list includes, for instance, the challenges of testing high coupled classes and of determining metrics for measuring the quality of the test suite. Results of the empirical studies indicate which and when strategies are faster and more effective on identifying faults in configurable software systems. Overall, we believe that practitioners acquire the necessary knowledge to choose a testing strategy that best fits their needs with our results and also benefit from our work, observing our solutions for each challenge.

Palavras-chave: Testing Configurable Systems, Testing Tools for Configurable Software Systems, Feature Interactions Faults.

List of Figures

1.1	Study steps	5
2.1	Feature model and some configurations of Weather Report	13
2.3	Feature interaction faults	18
2.4	Feature model and suggested configurations using a 1-, 2-, 3-, and 4-wise strategies for the ELEVATOR configurable system.	20
3.1	Filtering process conducted for study selection	28
3.2	Number of testing tools by publication venues	30
3.3	Tools distribution over years and testing strategies	31
3.4	Distribution of downloaded and tested tools	31
3.5	Distribution of testing strategy tools	35
3.6	Evaluation metrics found	39
3.7	Evaluation approaches found	40
4.1	Dataset creation process	45
5.1	Steps of the empirical study	54
5.2	Select of testing tools	55
5.3	Effectiveness data collection process	57
6.1	Time to run and to generate the list of configurations	71
6.2	Summary of t-wise strategies comparison	84
7.1	Faults distribution found	98

List of Tables

3.1	Studies obtained after the search process	29
3.2	Characteristics each tool	34
3.3	Testing strategies description	36
3.4	Testing strategies	37
3.5	Evaluation configurable systems found	39
4.1	Dataset metrics, adapted from Metrics [2020]	49
4.2	Size and variability metrics of the dataset	50
4.3	Test suite metrics of the dataset	51
5.1	Execution time	58
5.2	Measurement of effectiveness	59
6.1	Total time in seconds spent by the target testing strategies	70
6.2	P-value for RQ1 (time to generate and to execute configurations)	72
6.3	Percentage of configurations analyzed by strategies	74
6.4	P-value results for pertence of configurations analyzed	75
6.5	Recall of testing strategies	77
6.6	P-value results for Recall of testing strategies	77
6.7	Time Efficiency of testing strategies	79
6.8	P-value results for Time Efficiency by the strategies	79
6.9	Coverage Efficiency of testing strategies	81
6.10	P-value results for Coverage Efficiency of testing strategies	81
7.1	Faults found by class with traditional metrics	91
7.2	Faults found by class with variability metrics	92
7.3	Faults found by feature	95

Contents

Acknowledgments	vi
Resumo	viii
Abstract	ix
List of Figures	x
List of Tables	xi
1 Introduction	1
1.1 Problem and Motivation	2
1.2 Goals	4
1.3 Method	5
1.4 Contributions and Publications	7
1.5 Results	8
1.6 Thesis Project Outline	10
2 Background and Related Work	12
2.1 Developing Configurable Systems	13
2.2 Feature Interaction Problem	17
2.3 Testing Configurable Systems	18
2.4 Related Work	21
2.5 Final Remarks	24
3 Systematic Mapping Study	25
3.1 Planning	26
3.2 Execution	27
3.3 Overview of Primary Studies	30
3.4 Data Analysis	32

3.5	Evaluation Overview of Testing Tools for Configurable Systems	38
3.6	Implications for Researchers and Practitioners	40
3.7	Threats to Validity	41
3.8	Final Remarks	42
4	A Test-enriched Dataset for Configurable Software Systems	44
4.1	Selecting Subject Systems	46
4.2	Generating Variability Encoding Systems	46
4.3	Creating Test Suite	47
4.4	Collecting Metrics	48
4.5	Test-enriched Configurable System Dataset	48
4.6	Threats to Validity	49
4.7	Final Remarks	52
5	Sound Testing Tools: A Comparative Study	53
5.1	Goal and Research Questions	54
5.2	Study Steps	54
5.3	Efficiency of the Tools	58
5.4	Effectiveness of the Tools	59
5.5	Threats to Validity	60
5.6	Final Remarks	61
6	Evaluating T-wise Testing	63
6.1	Goal and Research Questions	64
6.2	Selected Testing Strategies	65
6.3	Data Acquisition	66
6.4	Operationalization	67
6.5	The Fastest Testing Strategies (RQ ₁)	69
6.6	The Most Comprehensive Strategies (RQ ₂)	73
6.7	The Most Effective Strategies (RQ ₃)	76
6.8	The Most Time-Efficient Strategy (RQ ₄)	78
6.9	The Most Coverage-Efficient Strategy (RQ ₅)	80
6.10	Grouping Testing Strategies	82
6.11	Implications for Practitioners	83
6.12	Implications for Researchers and Tool Builders	85
6.13	Threats to Validity	86
6.14	Final Remarks	87

7	Investigating the Dispersion of Faults over Classes and Features	88
7.1	Goal and Research Questions	89
7.2	Data Acquisition	89
7.3	Operationalization	90
7.4	Dispersion of Faults over Classes (RQ ₁)	91
7.5	Dispersion of Faults over Features (RQ ₂)	94
7.6	On the Relation between Faulty Classes and Faulty Features	96
7.7	Threats to Validity	97
7.8	Final Remarks	99
8	Ten Challenges for Testing Configurable Software Systems	101
8.1	Creating from Scratch and Expanding Test Suites	102
8.2	Creating Test Cases in Highly Coupled Classes	104
8.3	Dealing with the Combinatorial Explosion of Configurations	105
8.4	Sampling Configurations for Test	105
8.5	Running the Test Suites	106
8.6	Assessing the Quality of the Automated Test Suites	107
8.7	Measuring the Test Suites	108
8.8	Dealing with False Positives from Tests	108
8.9	Tracking Feature Interaction Faults to their Sources	109
8.10	Finding Technical Debts in Test Cases of Configurable Systems	110
8.11	Final Remarks	110
9	Conclusion	112
9.1	Summary	112
9.2	Contribution	116
9.3	Future Work	117
	Bibliography	118

Chapter 1

Introduction

Configurable software systems (for short *configurable systems*) are software systems that can be adapted based on a set of features that fits specific customer needs [Svahnberg et al., 2005; Pohl and Metzger, 2006; Apel et al., 2013a]. Features allow configurable systems to have an optional or incremental unit of functionality [Batory, 2005; Apel and Kästner, 2009]. Through variability, configurable systems can be efficiently extended, changed, customized or configured, according to different customer requirements [Svahnberg et al., 2005]. Nowadays, it is widespread the use of configurable systems. For instance, Web browser Firefox [Garvin and Cohen, 2011; Mozilla, 2020], Linux Kernel [Linux, 2020] and Android family [Galindo et al., 2016; Li et al., 2016] are well-known examples of such systems, with many variants for different users.

In practice, variations in configurable systems are achieved by two main implementation approaches [Apel et al., 2013a]: compile-time or execution-time. In the former [Kastner., 2010; Apel et al., 2013a; Czarnecki and Eisenecker, 2000], the code is annotated with `#ifdef`-like directives that check if the configuration options are enabled in order to generate the final product, for instance, by using conditional compilation. For instance, the Linux Kernel is implemented using this approach. In the latter [Post and Sinz, 2008; Apel et al., 2013c,a], the configuration options are represented through variables that guard code blocks which are enabled at execution time. For example, the Android family configurations options are implemented this way. Not limited to these two practices, recent studies have empirically shown the use of modularity-based approaches [Figueiredo et al., 2008; Gaia et al., 2014; Diniz et al., 2017; Cavarlé et al., 2018].

To address a diversity of deployment contexts and usages, developers of configurable systems only need to activate or deactivate features. Although configurable systems are expected to increase code reuse and productivity, developers have to deal

with several configuration options [Pohl et al., 2005]. Hence, to ensure that all configurations correctly compile, build, and run, developers spend considerable effort testing their systems. This effort is necessary mainly because configurations that fail may hurt potential users and degrade the reputation of a project [Halin et al., 2019].

Aware that software testing is a key component for ensuring that all configurations work properly, researchers have proposed testing strategies and methods [Classen et al., 2013; Henard et al., 2014b; Sánchez et al., 2014; Medeiros et al., 2016; Halin et al., 2019] as well as created tools [Kim et al., 2006; Hervieu et al., 2011; Henard et al., 2013a] and performed studies about software testing on configurable systems [Engström and Runeson, 2011; Lee et al., 2012b; Machado et al., 2014; Lopez-Herrejon et al., 2015]. Despite the number of studies on testing configurable systems, previous work [Engström and Runeson, 2011; Machado et al., 2014; Lopez-Herrejon et al., 2015] reports a lack of empirical evaluations including a community-wide dataset to guide the comparison of different testing strategies and tools.

1.1 Problem and Motivation

The recurring challenge in testing configurable systems is how to deal with a combinatorial explosion of configurations and tests [Apel et al., 2013a]. Although configurable systems may increase code reuse and productivity, they also have several configuration options and combinations of them to deal with. Therefore, testing configurable systems is more challenging than testing traditional systems. While in traditional software systems, there is only one product/configuration (combination of features), there are many configurations to run all tests in configurable systems, leading to a combinatorial explosion of configurations and tests [Apel et al., 2013a]. Therefore, testing thoroughly, against all configurations, is a costly practice. Alternatively, a popular strategy used in industry is to run the tests for a default configuration. This approach is efficient, but it can miss bugs [Greiler et al., 2012; Machado et al., 2014].

Between those two extremes, several approaches for testing configurable systems have been proposed [Cohen et al., 2003; Kuhn et al., 2004, 2010; Kim et al., 2012b, 2013; Liebig et al., 2013a; Nguyen et al., 2014; Medeiros et al., 2016; Souto et al., 2017]. Other approaches [Kim et al., 2013; Liebig et al., 2013a; Nguyen et al., 2014; Souto et al., 2017] take the code (test or system) into account in addition to the feature model, and dynamically explore all reachable configurations from a given test. Alternatively, developers may test a sample of valid configurations [Machado et al., 2014].

Configurable systems are very different from each other. For example, they can vary in the numbers of valid configurations, points of variability, size in lines of code, number of features, and functionality. To achieve desired levels of efficiency and effectiveness and given the peculiarities of each configurable system, choosing a more suitable test strategy is challenge. For example, a configurable system with thousands of valid configurations when using a sampling test strategy, that prioritizes some units of configurations for testing, will not check thousands of combinations of features. Even though the configurations returned by the strategy are representative, many faults may not be detected. However, if the tester wants to use a quick strategy for an initial check, it is not feasible to use a strategy that would cost hours to test thousands of configurations. In fact, we lack an in-depth study of configurable system testing that discusses and lists characteristics of configurable systems, characteristics of interactions of failing features, and characteristics of testing strategies. A study in this direction can support developers, testers, and researchers to test configurable systems.

Moreover, feature interaction faults are a significant challenge for configurable systems. Feature interactions occur when features influence the behavior of other features [Apel and Kästner, 2009; Apel et al., 2011]. It becomes an issue when developers look at features together and find an unexpected behavior that does not occur when they look at features in isolation [Soares et al., 2018b]. Unexpected behavior can introduce faults that manifest themselves in specific configurations. Feature interactions are among the greatest challenges in developing configurable systems [Kim et al., 2010; Apel et al., 2011; Garvin and Cohen, 2011; Siegmund et al., 2012; Abal et al., 2014; Machado et al., 2014; Schuster et al., 2014; Soares et al., 2018b; Nguyen et al., 2019] and enforces the need to create testing suites that cover all potential interactions [Cohen et al., 2008; Oster et al., 2011].

However, a prior work reports the lack of study on common faults on configurable systems, mainly emerged due to feature interactions [Machado et al., 2014]. Knowing common faults is an opportunity for researchers characterizing and providing recommendations for practitioners correct them as well as compare issues (e.g., bugs, faults, and variability-aware smells) that bother developers on the configurable system evolution and maintenance processes [Vale et al., 2014, 2015a]. Regarding practitioners, a deep understanding of feature interaction faults in configurable systems may help them to identify the reasons of faults that occur in their systems. Knowing the origin of faults may be useful to avoid the future appearance of the same failure and end with a vicious cycle of solving related faults. Practitioners may also use this knowledge to improve existing testing strategies.

Previous studies [Engström and Runeson, 2011; Machado et al., 2014; Lopez-Herrejon et al., 2015] have reported and created repositories of open-source configurable systems, although they neglected their test suites. However, no study comparing testing strategies with a community-wide dataset was found. While the comparison of testing strategies may benefit practitioners supporting their choice of a testing strategy that best fits their needs, a deep understanding of faults may help practitioners learn characteristics of classes and features prone to fail. Furthermore, developers can avoid similar faults, and testers can increase the test coverage in these fault-prone classes and features. On the other hand, comparing testing strategies with a community-wide dataset may also benefit researchers and tool builders by showing them opportunities for improving existing testing strategies and tools.

1.2 Goals

This thesis aims to provide a detailed view of testing strategies for configurable software systems. Besides, this work relies on this knowledge to propose a list of challenges faced when testing configurable systems and dealing with a test suite for our dataset systems. Our results can be seen as lessons learned on creating tests for configurable systems and they aim at supporting researchers and practitioners on this activity for configurable systems. Researchers may also use this knowledge to improve existing testing strategies. To achieve the general goals of this thesis, the following specific goals (SG) are defined.

- *SG1* Investigate testing tools and strategies for configurable systems in the literature.
- *SG2* Investigate which configurable systems are available in the literature to create a dataset of test-enriched configurable systems.
- *SG3* Perform comparative study with sound testing strategies.
- *SG4* Perform comparative study with t-wise testing strategies.
- *SG5* Analyze the dispersion of faults over classes and features in the subject dataset.
- *SG6* Propose a list of challenges faced when creating test suites for configurable systems and dealing with a test suite for our dataset systems.

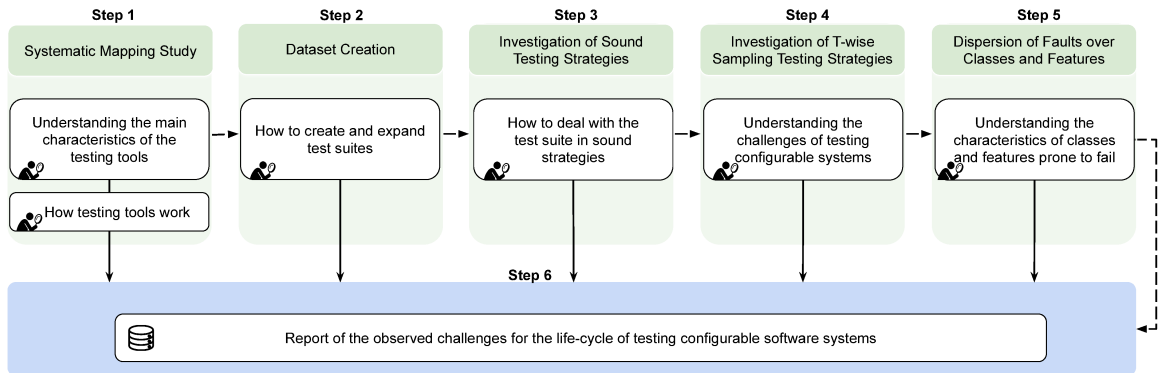


Figure 1.1: Study steps

1.3 Method

Considering that the lack of studies reporting and investigating the challenges on the complete life-cycle of testing configurable systems, we aim at filling this gap by **reporting the challenges we faced when creating, extending, assessing, using, and identifying faults on the test suites of 30 configurable systems**. These challenges can benefit both researchers and practitioners since the creation, use, and maintenance of test suites are a fundamental step for testing configurable systems and our lessons learned can be used, for instance, as a starting point for future research.

Figure 1.1 provides an overview of the steps of our research, highlighting the lessons learned at each step. We also use the knowledge on creating test cases to propose the ten challenges in this thesis. To achieve our goal, we performed a systematic mapping study (Step 1). Next, we created and/or extended test suites for 30 configurable systems (Step 2). Then, we empirically investigated testing strategies to generate a set of configurations to be tested (Step 3 and Step 4). In Step 5, we look for understanding the characteristics of classes and features prone to fail. Finally, we summarize the observed challenges during the life-cycle of testing configurable systems (Step 6).

Step 1. We performed a systematic mapping study aiming at identifying the existing testing tools for configurable systems (see Chapter 3). In this systematic mapping study, we (i) provided an overview of the state-of-the-art of testing strategies for configurable systems, (ii) analyzed the main characteristics of the testing tools for configurable systems, and (iii) presented the main strategies used to test configurable systems. We found a total of 60 tools and collected the main characteristics of testing tools for configurable systems. Through this step, we understand how the testing tools for configurable systems works and their main characteristics. In addition, Step 1

was essential to discover what are the main restrictions of testing configurable systems (Chapter 3).

Step 2. We conduct an *ad hoc* literature review by analyzing survey papers on testing configurable systems and well-known datasets of configurable systems (e.g., SPL2GO¹ and SPL REPOSITORY²) (see Chapter 4). Despite we have found 60 configurable systems developed in JAVA-based programming languages, only 10 configurable systems have their test suite public available. Aiming at increasing this number, we create a test suite for further 20 systems. The creation of tests followed two stop criteria: (i) at least 70% of code coverage and (ii) at least 40% of the mutants killed. These two criteria aim at providing a high-quality testing dataset. At the end, we created test suites for 30 configurable systems. With the creation and expansion of the test suites for the dataset systems, we identified the main challenges faced. This dataset has been used by the community to research testing approaches in configurable systems [Ferreira et al., 2020d].

Step 3. We conducted a comparative study (see Chapter 5) to evaluate the two main sound tools found in the systematic mapping studies: *VarexJ* [Meinicke et al., 2016] and *SPLat* [Kim et al., 2013]. The main goal of that step was to identify advantages and drawbacks on testing tools that use sound testing techniques. Hence, we measured the effectiveness and efficiency of *VarexJ* and *SPLat* to test seven systems identified in Step 2. We observed that, even though both tools claim to use sound testing technique strategy, they generally did not present high intersection rates. Alternatively, our experiments have shown that for configurable systems with up to 17 features, it was possible to use testing strategies that test all configurations.

Step 4. In this study (see Chapter 6), we compared variations of five t-wise sampling testing strategies, named *CASA* [Garvin et al., 2011], *Chvatal* [Johansen et al., 2012b], *ICPL* [Johansen et al., 2012b], *IncLing* [Al-Hajjaji et al., 2016a], and *YASA* [Krieter et al., 2020]. We selected t-wise strategies because they ensure a degree of coverage for determining the set of configurations recommended by the strategy. At the end, we compared sixteen t-wise strategies (*CASA-T1*, *CASA-T2*, *CASA-T3*, *CASA-T4*, *Chvatal-T1*, *Chvatal-T2*, *Chvatal-T3*, *Chvatal-T4*, *ICPL-T1*, *ICPL-T2*, *ICPL-T3*, *IncLing-T2*, *YASA-T1*, *YASA-T2*, *YASA-T3*, and *YASA-T4*) and two base-lines (*brute force* and *random selection*).

Step 5. We investigated the dispersion of faults over classes and features from the dataset (see Chapter 7). A deep understanding of faults may help practitioners to learn characteristics of classes and features prone to fail, to avoid the introduction of similar

¹<http://spl2go.cs.ovgu.de/>

²http://labsoft.dcc.ufmg.br/doku.php?id=%20about:spl_list

faults, and to guide them to increase the test coverage in these fault-prone classes and features. We first retrieve a list of all classes that failed and discuss their dispersion over each subject configurable system. Then, aiming at discovering whether these faulty classes have distinct characteristics from other classes, we compute traditional and CK metrics for each class of each subject system. Similar to investigating characteristics of faulty classes, we investigate characteristics of faulty features. We use metrics related to features, such as the number of classes and methods a feature is located (scattering).

Step 6. In this step, we summarize the main challenges observed in the previous steps. Although previous work concentrates on *the explosion of combinations* [Apel et al., 2013a] and *feature interactions* [Apel and Kästner, 2009; Apel et al., 2011] challenges, we present other challenges faced when testing configuration systems in practice. For instance, the challenges (1) on the creation of a test suite, (2) on measuring the test suite and its quality, and (3) on the identification of faults are often ignored. We defined ten challenges related to configurable software testing based through our experience when creating and extending the test suite for our dataset systems (Chapter 4). In this step, our main goal is to report the main challenges in the complete life-cycle of 30 open-source configuration systems. Therefore, instead of focus on the well-known challenges present in the literature, we present challenges faced when creating, extending, assessing, and using test suites for 30 configurable systems.

1.4 Contributions and Publications

This thesis main contribution is the empirical knowledge about testing configurable systems. We believe that with our results, practitioners acquire the necessary knowledge to choose a testing strategy that best fits their needs. Moreover, researchers and tool builders are served with opportunities to improve existing testing strategies and tools. We also observed a lack of information on creating tests for configurable software systems. We propose a list of ten challenges faced when performing test suites for configurable systems and dealing with a test suite for our dataset systems. Our list includes, for instance, the challenges of testing high coupled classes and of determining metrics for measuring the quality of the test suite. Our results can be seen as lessons learned on creating tests for configurable systems and they aim at supporting researchers and practitioners on this activity for configurable systems. Until now, the research reported in this thesis has generated the following publications:

1. Ferreira, Fischer; Vale, Gustavo; Diniz, João Paulo; Figueiredo, Eduardo; 2021. Evaluating T-wise Testing Strategies in a Community-wide Dataset of Configurable Software Systems. *Journal of Systems and Software (JSS)*, <https://doi.org/10.1016/j.jss.2021.110990>.
2. Ferreira, Fischer; Diniz, João Paulo; Vale, Gustavo; Figueiredo, Eduardo; 2021. On the Challenges for Creating a Test Suite for Configurable Software Systems. *Proceedings of the 24th Ibero-American Conference on Software Engineering, Software Engineering Track (CIbSE - SET)*.
3. Ferreira, Fischer; Vale, Gustavo; Diniz, João Paulo; Figueiredo, Eduardo; 2020. On the Proposal and Evaluation of a Test-enriched Dataset for Configurable Systems. *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*.
4. Ferreira, Fischer; Viggiato, Markos; Souza, Maurício; Figueiredo, Eduardo; 2020. Testing Configurable Software Systems: The Failure Observation Challenge. *Proceedings of the 24th ACM International Systems and Software Product Line Conference (SPLC)*.
5. Ferreira, Fischer; Figueiredo, Eduardo; 2020. A Test Strategy for Configurable Software Systems using Machine Learning. *Proceedings of the 23th Ibero-American Conference on Software Engineering (CIbSE) Doctoral Symposium*.
6. Ferreira, Fischer; Diniz, João Paulo; Silva, Cleiton; Figueiredo, Eduardo; 2019. Testing Tools for Configurable Software Systems: A Review-based Empirical Study. *Proceedings of the 13th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*.

1.5 Results

This thesis provides the following main results.

We found 25 primary studies cited in previous secondary studies [Meinicke et al., 2014; Machado et al., 2014; Lopez-Herrejon et al., 2015; Varshosaz et al., 2018] and another 39 primary studies on testing tools for configurable systems. Our mapping study covered studies published between 2014 and 2020. Therefore, in total, we found 64 papers resulting in 60 testing tools for configurable systems. We analyzed the tools found concerning 16 characteristics and four main testing strategies. This thesis shows an overview of 64 primary studies found and presents an overview of how the

researchers evaluated testing tools for configurable systems. Finally, we discuss our results regarding implications for researchers and practitioners. We have also listed all papers used in the proposed mapping study steps [Ferreira et al., 2020a].

We searched for configurable systems in the literature and we found 243 systems being 60 developed in a Java-based programming language. However, only ten systems have a test suite available. In this way, we created tests until they have a coverage of 70 % and kill at least 40 % of mutants for the other 20 projects. These two criteria aim at providing a high-quality testing dataset. At the end, we created a dataset with 30 test-enriched configurable systems varying in domains, size, variability, and test suite size. We make our infrastructure and data publicly available for follow-up studies on a supplementary website [Ferreira et al., 2020c].

We designed and executed a comparative study with two sound testing strategies, namely *SPLat* and *VarexJ*. In this empirical study, we note that *VarexJ* is generally more efficient than *SPLat*. However, when it was not more efficient, it was by a large difference in specific situations related to its implementation of variability-aware execution. We observed that *VarexJ* and *SPLat* presented different results for efficiency while testing the target systems and that, although *VarexJ* found more faults than *SPLat* for the majority of the target systems, such result deserves a more in-depth investigation because we expected a higher intersection of faults encountered by them.

In the second empirical study, we compare the performance of the configurations suggested by sixteen t-wise strategies. We found that for each t-wise group: (i) *ICPL-T1*, *ICPL-T2*, *ICPL-T3* and *YASA-T4* are usually *fast* in relation to groups 1-, 2-, 3-, and 4-wise, respectively; (ii) *Chvatal-T1*, *IncLing-T2*, *Chvatal-T3*, and *Chvatal-T4* are the most *comprehensive* testing strategy; (iii) *Chvatal-T1*, *ICPL-T2*, *Chvatal-T3*, and *Chvatal-T4* are the testing strategies that recommends configurations able to find the *greatest number of faults*; (iv) *Chvatal-T1*, *ICPL-T2*, *Chvatal-T3*, and *YASA-T4* are the testing strategy that recommends configurations able to find the *best balance among faults found and time*; and (v) *Chvatal-T1*, *CASA-T2*, *Chvatal-T3*, and *Chvatal-T4* are the testing strategies that recommends configurations able to find the *best balance among faults found and the number of recommended configurations* in relation to groups 1-, 2-, 3-, and 4-wise.

In the third empirical study, we look at the dispersion of faults found over classes and features in the subject dataset. Then, we measure each class and feature with metrics commonly used in practice [Chidamber et al., 1998]. Number of lines of code (*LoC*) [CK, 2020], weighted methods per class (*WMC*) [CK, 2020], and response for a class (*RFC*) [CK, 2020] are examples of metrics at class-level. Feature scattering and feature tangling are examples of feature-level metrics. Finally, we compute Spearman's

rank correlation between the number of faults in a component (i.e., class or feature) and a given metric aiming to provide a deep understanding of faults found in a community-wide dataset. We found that faults are usually concentrated in a few classes and features and these fault-prone components are distinguishable from components safe of faults only measuring their source code with metrics normally used in practice.

Finally, in this thesis we discussed ten challenges related to the life-cycle of tests for configurable systems. We summarized the challenges through our observations based on the background acquired from previous empirical studies. We believe that researchers and practitioners can benefit from the described challenges and can propose solutions for them making our challenges just a starting point for future research. Therefore, instead of focus on the well-known challenges present in the literature, we present challenges faced when creating, extending, assessing, and using test suites for 30 configurable systems. For brief, our list of challenges is:

1. *Creating from scratch and expanding test suites*
2. *Creating test cases in highly coupled classes*
3. *Dealing with the combinatorial explosion of configurations*
4. *Sampling configurations for test*
5. *Running the test suites*
6. *Assessing the quality of the automated test suites*
7. *Measuring the test suites*
8. *Dealing with false positives from tests*
9. *Tracking feature interaction faults*
10. *Finding technical debts in test cases of configurable systems*

1.6 Thesis Project Outline

In addition to this introductory chapter, the remainder of this thesis project is organized as follows.

Chapter 2 provides the essential concepts to support this thesis. In addition to details concerning testing of configurable systems. We present an overview of variability

encoding, techniques, and challenges related to develop configurable systems. This chapter also discusses related work.

Chapter 3 presents the protocol and results of a systematic mapping study conducted to identify the state-of-the-art testing tools for configurable systems.

Chapter 4 presents the proposal of a test-enriched dataset to support the evaluation of testing strategies for configurable systems. The proposed dataset is the first dataset for configurable systems with extensive test suites.

Chapter 5 compares two sound testing tools called *VarexJ* and *SPLat*. We found these test tools in the systematic mapping study (Chapter 3) .

Chapter 6 presents a comparison of variations of five t-wise sampling testing strategies *CASA* [Garvin et al., 2011], *Chvatal* [Johansen et al., 2012b], *ICPL* [Johansen et al., 2011], *IncLing* [Al-Hajjaji et al., 2016a], and *YASA* [Krieter et al., 2020]. We found these five t-wise sampling testing strategies in the systematic mapping study (Chapter 3).

Chapter 7 presents at the dispersion of faults found over classes and features in the subject dataset.

Chapter 8 presents a list of ten challenges faced when performing test suites for configurable systems and dealing with a test suite for our dataset systems.

Chapter 9 presents the conclusion of this thesis, reviewing the results concerning the specific goals. We discuss the contributions and implications of this thesis. We also discuss future works as a consequence of this study.

Chapter 2

Background and Related Work

Software testing is a key component for ensuring that all configurations work properly [McGregor, 2001; Ammann and Offutt, 2016]. Despite its importance, testing configurable software systems is more challenging than testing traditional software systems. While in traditional software systems there is only one product or configuration (a combination of features) to be tested, for configurable systems, we need to run all tests in several different configurations, which makes exhaustive testing prohibitively expensive and practically infeasible [Apel et al., 2013a].

Alternatively, developers may test a sample of valid configurations [Machado et al., 2014]. Several strategies for choosing a sample of configurations have been proposed [Medeiros et al., 2016; Souto et al., 2017]. Some of them use information only available in the *feature model* [Nie and Leung, 2011; Johansen et al., 2011; Al-Hajjaji et al., 2016a; Medeiros et al., 2016], while others also use information from the source code [Kim et al., 2013; Nguyen et al., 2014; Souto et al., 2017].

This chapter presents an overview of approaches and techniques to develop configurable systems. The remainder of this chapter is organized as follows. Sections 2.1 and 2.2 provide an overview of developing configurable systems and feature interaction problem, respectively. Section 2.3 shows an overview of testing configurable systems and t-wise techniques. Section 2.4 discusses the related work. Finally, Section 2.5 concludes this chapter.

2.1 Developing Configurable Systems

Configurable software systems are software systems that can be adapted or configured according to a set of features (configuration options) [CLEMENTS and Northrop, 2002]. A feature is a unit of functionality of a configurable system that satisfies a requirement, represents a design decision, and provides a potential configuration option [Apel and Kästner, 2009]. Features allow configurable systems to have an optional or incremental unit of functionality [Batory, 2005; Cruz et al., 2019]. Through variability, configurable systems can be extended, changed, customized or configured, according to different customer requirements [Svahnberg et al., 2005]. Nowadays, it is widespread the use of configurable systems. For instance, Web browser Firefox [Garvin and Cohen, 2011; Mozilla, 2020], Linux Kernel [Linux, 2020], and Android family [Galindo et al., 2016; Li et al., 2016] are well-known examples of such systems, with many variants for different users.

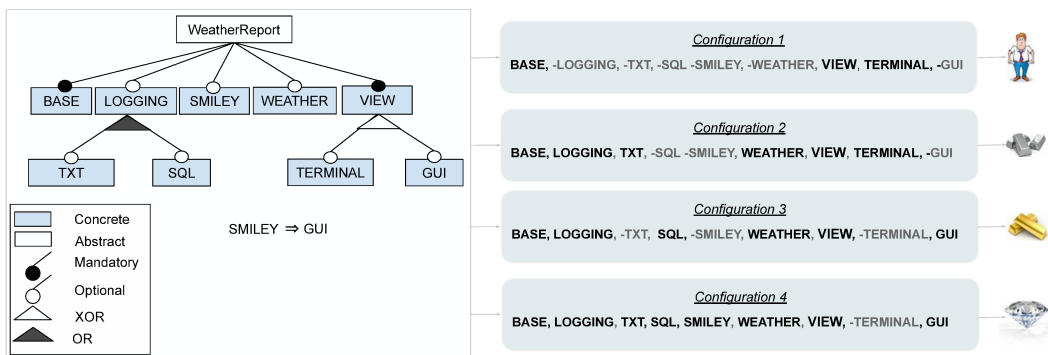


Figure 2.1: Feature model and some configurations of Weather Report

Configurable software systems offer numerous options (or features) to fit specific customer needs [Svahnberg et al., 2005; Pohl and Metzger, 2006; Apel et al., 2013a]. This way, developers may activate or deactivate options to address a diversity of deployment contexts and usages. Figure 2.1 shows an example of the configurable system called WEATHER REPORT. This configurable system has nine concrete features (e.g., SMILEY and WEATHER) and four examples of configuration options. A configuration represents a selection of features from a feature model [Krzysztof and Eisenecker, 2000]. The feature model is a way to represent variability in configurable systems, as it is the most popular model [Riebisch, 2003] this thesis provides an overview of this form of representation. A feature model is a tree structure that capture the informa-

tion of all the possible configurations of the configurable system in terms of features and relationships among them [Krzysztof and Eisenecker, 2000]. As an example, the configurable system WEATHER REPORT has 24 valid configurations.

Configuration 1 (Figure 2.1) represents a version of WEATHER REPORT with only basic functionalities. In this case, only mandatory features are enabled (BASE, and VIEW). If a child feature is mandatory, it is enable in all configurations in which its parents are enable. In Configuration 2, in addition to the mandatory features, the features LOGGING, TXT, WEATHER, and TERMINAL, are enabled, allowing WEATHER REPORT to have other functionalities. The features GUI, LOGGING, SMILEY, and WEATHER are optional features. If a child feature is defined as optional, it can be optionally enabled in configurations in which its parents are enabled.

In a feature model, it is possible to define the features as XOR alternatives and OR. A set of child features is defined as an alternative feature if only one feature of this set can be selected when its parents are enabled in a configuration. Figure 2.1 presents two alternatives features (TERMINAL and GUI). As the feature VIEW is mandatory, each configuration must contain either the features TERMINAL OR GUI and a configuration cannot have these features enabled at the same time. Regarding OR-relation features: a set of child features is said to have an or-relation with their parents when one or more of them can be enable in the configuration in which their parents are enable. Figure 2.1 presents two or-relation features (TXT and SQL). If feature LOGGING is enable in a configuration at least one of the features TXT and SQL must be enable. The propositional formulas under the tree represents the cross-tree constraints. For example, in Figure 2.1 if the SMILEY feature is enable the GUI feature has to be enable. In this way, several versions of WEATHER REPORT can be generated according to the feature model.

To develop configurable systems, developers may choose between the two main strategies: *compile-* and *execution-time* [Apel et al., 2013a]. In compile-time strategies, developers activate a set of features that is (pre)processed in order to generate the final product [Czarnecki and Eisenecker, 2000; Kastner., 2010; Apel et al., 2013a]. This strategy can be divided into annotative and compositional approaches. The main difference among these approaches is that in the annotative strategy, the developers annotate their code (e.g., with `#ifdef`-like directives) to represent variation points. However, developers implement each variation point in modularized components in compositional approaches (e.g., features, aspects or deltas).

Compositional approaches. In compositional approaches, developers implement each variation point in modularized components (e.g., features [Batory, 2005; Ferreira et al., 2014] to Feature-oriented programming, aspects [Kiczales, 1996; Figueiredo et al., 2008; Schaefer et al., 2011] to AOP or deltas [Schaefer et al., 2010; Diniz et al., 2017] to DOP). Conditional compilation (CC) is an example of a technique that uses the annotative approach [Post and Sinz, 2008; Apel et al., 2013c]. Feature-oriented programming (FOP) [Batory, 2005], aspectual feature modules (AFM) [Apel et al., 2008; Gaia et al., 2014], and delta-oriented programming (DOP) [Schaefer et al., 2010] are examples of techniques that use a compositional approach. The LINUX KERNEL [Linux, 2020] and the FIREFOX WEB BROWSER [Garvin and Cohen, 2011; Mozilla, 2020] are examples of configurable systems developed with compile-time strategies.

Variability Encoding. In execution-time strategies, developers activate features that contain code blocks at execution-time [Post and Sinz, 2008]. Variability encoding is an example of execution-time strategy [Apel et al., 2013a,c]. Similar to conditional compilation, developers should create conditional structures (e.g., `if/else` statements or ternary operator `?:`) and generate the so-called *meta-products* [Thüm et al., 2012] in variability encoding. Then, developers should create configuration files determining features to be enabled in a target configuration. This way, all features can be activated or deactivated at execution time [Kim et al., 2013; Meinicke et al., 2016; Wong et al., 2018]. ANDROID FAMILY is a (set of) configurable system(s) developed using variability encoding [Galindo et al., 2016; Li et al., 2016]. In this study, configurable systems are implemented using variability encoding and, through manipulating functional features, they are added at run-time. We do not consider systems in which program parameters [Apel et al., 2013a] are used to determine configurations.

Variability Encoding is the technique used for our dataset systems presented in Chapter 4. Configurable software systems have long been studied by the software product line engineering community [Pohl and Metzger, 2006; Apel et al., 2013b]. Among the strategies to introduce variability in software systems, variability encoding has drawn practitioners attention since developers only need to *annotate variation points* on their existing systems. This way, developers simply activate or deactivate features to address different deployment contexts. For short, while annotating variation points, developers should create a configuration file where they determine options that are going to be enabled in a target variation.

Listing 2.1 presents a fragment of code in a configurable system of our dataset (Chapter 4), named *Companies*. In this example, the method `getTotal` returns a string containing a calculated value. If the feature `TOTAL_WALKER` is enable, the method returns the value calculated by the `TOTALWALKER` class. If the fea-

ture `TOTAL_REDUCER` is enable, the method returns the value calculated by the `TOTALREDUCER` class. On the other hand, if the features `TOTAL_WALKER` and `TOTAL_REDUCER` are not enable, no value is calculated. The variations in configurable systems of our dataset (4) use the variability encoding (execution-time) technique.

```

1 public String getTotal() {
2     String value = "";
3     if (Configuration.TOTAL_WALKER) {
4         TotalWalker walker = new TotalWalker();
5         walker.postorder(currentValue);
6         value = Double.toString(walker.getTotal());
7     } else if (Configuration.TOTAL_REDUCER) {
8         TotalReducer total = new TotalReducer();
9         double valueDouble = total.reduce(currentValue);
10        value = Double.toString(valueDouble);
11    }
12    return value;
13 }

```

Listing 2.1: Variability encoding example

To translate a configuration systems code from Annotative to Variability Encoding format, we use the process described by Souto [2015]. We demonstrate a common example proposed in her study and applied in our thesis (listings 2.2 and 2.3). Listing 2.2 presents an Annotative example and Listing 2.3 presents their translation to Variability Encoding. We need to guard the definitions and uses of the Annotative translating into Variability Encoding remaining the same behavior.

```

1 class ClassExample {
2     void methodNumber(){
3         int x = 0;
4         //#ifdef A
5             x = 10;
6         //#endif
7         //...
8         //#ifdef B
9             x = 20;
10        //#endif
11        //...
12    }
13 }

```

Listing (2.2) Annotative example

```

1 class ClassExample {
2     void methodNumber(){
3         int x = 0;
4         if(Configuration.A)
5             x = 10;
6
7         //...
8         if(Configuration.B)
9             x = 20;
10
11        //...
12    }
13 }

```

Listing (2.3) Variability encoding example

2.2 Feature Interaction Problem

The concept of feature interactions was initially studied in telecommunication systems [Bowen et al., 1989]. The first reported feature interaction problem was unexpected behavior of the system in case of concurrent features: call waiting, and call forwarding. If both are active, the system can behave in a non-deterministic way, sometimes putting the call on hold, occasionally forwarding the call to another number (or mailbox).

Feature interactions occur when features influence the behavior of other features [Batory et al., 2011]. It becomes an issue when developers look at features together and find an unexpected behavior that does not occur when they look at features in isolation [Soares et al., 2018b]. Unexpected behavior can introduce faults that manifest themselves in specific configurations. Feature interaction faults are among the greatest challenges in developing configurable systems [Kim et al., 2010; Apel et al., 2011; Garvin and Cohen, 2011; Siegmund et al., 2012; Machado et al., 2014; Schuster et al., 2014; Soares et al., 2018b; Nguyen et al., 2019] and enforces the need to create testing suites that cover all potential interactions [Cohen et al., 2008; Oster et al., 2011]. These faults are usually caused by a problem with the interactions of two or more features.

```
1 public class WeatherReport {
2     private Date date;
3     private String temperature;
4     public WeatherReport(Date currentDate, String currentTemperature){
5         this.date = currentDate;
6         this.temperature = currentTemperature;
7     }
8     public String createText(String c) {
9         if (Configuration.SMILEY)
10            c = c.replace(":", getSmiley(":"));
11        if (Configuration.WEATHER)
12            c = c.replace("[:weather:]", temperature);
13        return c;
14    }
15 }
```

Listing 2.4: Variability encoding example adapted from Meinicke et al. [2016]

To illustrate, Listing 2.1 presents a traditional and simple example of feature interaction problem adapted from WordPress [Meinicke et al., 2016]. In this example, the *createText* method edits and returns a string *c*. If only feature SMILEY is enable,

the method replaces `:/` by `☺`. Looking at Listing 8.1, we see that both features use a string `“:/”`. This fact may cause a feature interaction fault. To illustrate, let us consider that `createText` method receives as input the string `“[:weather:]”` (i.e., `c = [:weather:]`) and both features `SMILEY` and `WEATHER` are enabled. This way feature `SMILEY` will change string `c` to `[:weather☺]` and the code of feature `WEATHER` will not find the expected string `c` anymore, even though it was exactly the expected input string before.

2.3 Testing Configurable Systems

A major challenge for developers of configurable software systems is to ensure that all configurations correctly compile, build, and run. Even when each feature performs well individually, interactions among them may happen and introduce unexpected behavior to the system [Apel et al., 2013b]. Feature interaction enforces the need for test suites covering all potential feature interactions. Figure 2.3 presents an example of four configurations of the `WEATHER REPORT` submitted to the test suite. We see that both `SMILEY` and `WEATHER` features are enabled in the `CONFIGURATION 4`. As a result, the test suite reports fault. However, the test suite does not find faults for `CONFIGURATIONS 1, 2, and 3`. We should note that above a certain amount of features, it is infeasible to test all possible feature combinations. This way, researchers and practitioners have to choose somehow the configurations they want to test.

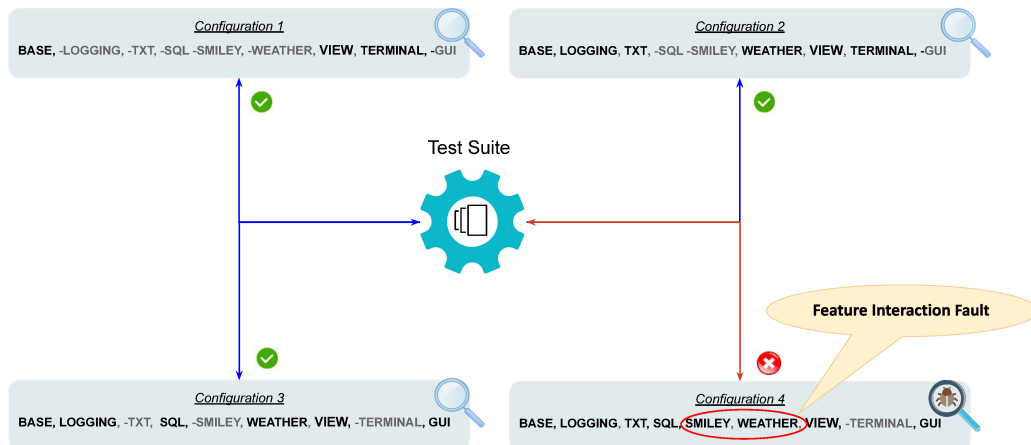


Figure 2.3: Feature interaction faults

Listing 2.5 demonstrates a test class for the example presented in Listing 2.2. In addition to the environment set up, there is a test case testing the `createText` method

when the feature WEATHER is enable. For short, receiving string “[:weather:]” it should return “30.0°C”.

```
1 public class WeatherReportTest {
2     WeatherReport wr;
3     @Before
4     public void setUp() {
5         WeatherReport wr = new WeatherReport("2020-08-25", "30.0°C");
6     }
7     @Test
8     public void weatherTest () {
9         if (Configuration.WEATHER)
10            assertEquals(wr.createText("[:weather:]"), "30.0°C");
11     }
12 }
```

Listing 2.5: Test cases for WeatherReport class

The fact that the number of product variants grows exponentially with the number of variation points makes it infeasible to test all possible feature combinations after a certain number of features. This way, practitioners have to choose somehow to test only a sample of configurations. Over the years, various strategies have been developed to test configurable systems [Engström and Runeson, 2011; da Mota et al., 2011; Machado et al., 2014; Ferreira et al., 2019]. These strategies can be classified into: variability-aware testing [Kim et al., 2013; Meinicke et al., 2016; Wong et al., 2018] and configuration sampling testing [Johansen et al., 2012a; Al-Hajjaji et al., 2016a; Souto et al., 2017]. *Variability-aware testing strategies* explore dynamically all reachable configurations from a given test, by monitoring feature variable accesses during test execution.

Configuration sampling. Configuration sampling testing strategies sample a subset of valid configurations and test them individually. We focus on configuration sampling testing strategies because they are more often used in practice [Varshosaz et al., 2018]. Configurable sampling testing strategies (for short, *sampling strategies*) can be classified into four groups [Varshosaz et al., 2018]: *manual selection*, *semi-automatic selection*, *automatic selection*, and *coverage*. In the first, practitioners should manually select the configurations to be tested. In the second, the selection of configurations requires an input representing the stop criteria (e.g., the number of products to be generated, the time for sampling, or a degree of coverage). In the third, the selection of configurations has support of greedy (i.e., optimal interactive choice) or meta-heuristic algorithms (e.g., local search or population-based search). In the fourth, the selection of configurations uses the coverage criteria to assure the quality of product sampling

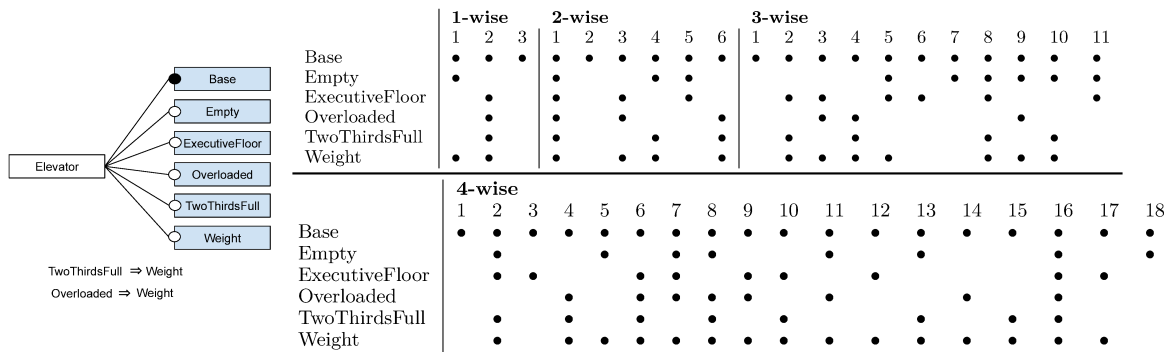


Figure 2.4: Feature model and suggested configurations using a 1-, 2-, 3-, and 4-wise strategies for the ELEVATOR configurable system.

(e.g., source code coverage or *feature interaction coverage*). From now on, we call this last group *t-wise* strategies.

Note that sampling strategies from one group may have the same goal of other groups and, for that reason, provide similar outcomes. For instance, a semi-automatic algorithm that covers all source code will provide a similar set of configurations that a coverage algorithm provides. Alternatively, a greedy automated algorithm that aims at selecting an optimal number of configurations that test all pairs of features individually, provides a similar set of configurations that a 2-wise algorithm provides [Lübke et al., 2019].

T-wise Techniques. T-wise techniques select a subset of configurations that covers a valid group of t features being activated and deactivated simultaneously [Henard et al., 2014a; Xiang et al., 2021]. This subset of configurations should respect constraints in the feature model to be called valid configurations. For instance, an 1-wise algorithm should select a set of configurations that all optional features are enable and deactivate at least once. On the other hand, a 2-wise algorithm (also known as pair-wise) should select a set of valid configurations that all pairs of optional features ($F1$ and $F2$) are simultaneously activate ($F1 \& F2$), alternatively activate ($F1 \& \sim F2$ and $\sim F1 \& F2$), and mutually deactivated ($\sim F1 \& \sim F2$). Note that the set of configurations may vary depending on the algorithm/strategy used.

To illustrate, consider the feature model of the ELEVATOR configurable system shown in the left-side of Figure 2.4. This system has six features: one mandatory (*Base*) and five optional (*Empty*, *ExecutiveFloor*, *Overloaded*, *TwoThirdsFull*, and *Weight*). The number of valid configurations is 20. In the right-side of Figure 2.4, we see the selected configurations to fulfill the requirements of 1-, 2-, 3-, and 4-wise using the *Chvatal* [Johansen et al., 2012b] strategy. For instance, to fulfill 1- and 2-wise requirements, this strategy selects three and six different configurations, respectively.

We used the *unit testing* and *mutation testing* techniques to create the test suite for the configurable systems presented in Chapter 4. *Unit testing*, Among techniques for testing software systems in general, unit testing are widely used [Jia and Harman, 2010; Papadakis et al., 2018a; Petrović and Ivanković, 2018; Papadakis et al., 2018b; Mao et al., 2019]. *Unit tests* focus the verification effort on the smallest software unit, such as a function or a software component. This way, important control paths are tested to find errors within component boundaries. The relative complexity of the tests and the found faults are limited by the scope established for a specific unit test, and unit tests usually have one or a few inputs and usually a single output.

Mutation testing, on the other hand, is a fault-based technique commonly used to evaluate the effectiveness of software testing [Gopinath et al., 2014; Just et al., 2014]. Even though it is a self cost technique, it has been empirically shown to be one of the most robust test selection criteria when compared to other measures such as control flow and data flow-based testing [Andrews et al., 2005; Gopinath et al., 2014; Just et al., 2014]. Mutation testing consists of introducing syntactical changes, called mutations, into the source code and check whether the test cases distinguish them [DeMillo et al., 1978]. The resulting programs are called mutants. A mutant is killed if the test suite observes the modification of the mutant from the original code. If the test suite cannot see changes, the mutant remains alive indicating that the test suite is not good enough to identify faults and developers should improve their test suite to kill surviving mutants.

2.4 Related Work

There are dozens of papers related to testing configurable systems [Engström and Runeson, 2011; Lamancha et al., 2013; Lee et al., 2012b; Lopez-Herrejon et al., 2015; Machado et al., 2014; da Mota et al., 2011; Pohl and Metzger, 2006]. We present studies that investigate faults, feature interactions, compare testing strategies for configurable systems, and variability bugs datasets.

Faults on Configurable Systems. Lopez-Herrejon et al. [2014] proposed a dataset with 19 configurable systems obtained through studies published in the five events (SPLC, VAMOS, ICSE, ASE, and FSE), and four repositories publicly available (SPL CONQUEROR, FEATUREHOUSE, SPL2GO, and SPLOT). Our thesis has three systems in common (ARGOUML-SPL, GPL, and ZIPME). Their work analyzed configurations recommended by *CASA*, *PGS*, and *ICPL* testing strategies in terms of size, performance, similarity, and frequency, considering information from the feature model.

The objective of their work is to evaluate the effectiveness of the testing approaches which is similar to one of our goals. In addition to having additional goals, in Chapter 6, we evaluate a larger set of systems, other testing strategies (*Chvatal* and *IncLing*), and other properties (eg , time, comprehensiveness, time-efficient, and coverage-efficient). In line with our results, *ICPL* was one of the fastest strategies.

Sánchez et al. [Sánchez et al., 2017] mined a list of faults from DRUPAL. DRUPAL is a modular web content management framework written in PHP with 48 features and 21 cross-tree constraints. They identified 3392 faults, and report feature interactions associated with these faults in two analyzed DRUPAL versions (v7.22 and v7.23). They characterized DRUPAL through the number of changes (during two years), cyclomatic complexity, number of test cases, number of test assertions, number of developers, and number of reported installations. Different from them, we use a much larger set of metrics and multiple systems. We present our dataset in Chapter 4. Our results are inline with their study concerning the number of code lines handled by the feature and weight method class can be used as good estimators for identifying fault-prone features and classes.

Fischer et al. [2018] proposed a dataset with six configurable systems for evaluating the fault detection capabilities of configurable systems testing strategies. Our dataset has two configurable systems in common (GPL and NOTEPAD) (Chapter 4). They reported that one of the main limitations of their work was the automatic generation of test cases using the EVOSUITE tool. Our dataset addresses this limitation since we manually developed a test suite for systems without one and extending the test suite of other systems that already had one. We believe that our test suites are more comprehensive than theirs, which reflects also on the number of faults we found. Similar to their work, we introduced mutations to emulate faults and verify the effectiveness of the test suite created (Chapter 4).

Feature Interaction Investigations. Considering that identify feature interaction faults is expensive and challenging, several studies in the literature have proposed different approaches to deal with the feature interaction [Garvin and Cohen, 2011; Machado et al., 2014; Schuster et al., 2014; Siegmund et al., 2012; Soares et al., 2018a,b]. As an example, Soares et al. [2018a] proposed a strategy called VARXPLOER, which focuses on pair-wise feature interactions. VARXPLOER is an incremental and interactive lightweight process to detect problematic interactions dynamically. Through VARXPLOER, the user identifies features that should not interact, and this way, a specification of the features is created. Our thesis also investigates feature interaction faults. However, we use the suite of automated and configurations recommended by sixteen t-wise strategies to inspect features (Chapter 6). Our results demonstrate that

it is possible to identify feature interaction problems through automated tests. The creation of a formal specification is costly, may be incomplete, ambiguous, or have errors. As an alternative to find feature interaction faults, automated testing can be a viable alternative.

Comparison of Testing Strategies Designed for Configurable Systems. [Medeiros et al., 2016] investigated faults using ten sampling algorithms (SPLCATool, CASA, ACTS, Statement-coverage, Most-enabled-disabled and Random) and 135 configuration-related faults from 24 subject systems developed in C programming language and ten different types of faults such as Memory Leaks. The main difference from our study is that we evaluated the configurations recommended by t-wise strategies in Java-based configurable systems and we provide analyses investigating the dispersion of faults on classes and features. Furthermore, while in their study, researchers manually created a corpus of 135 faults, we identified faults automatically. Regarding the difference on results, Medeiros et al. [2016] showed that the recommended configurations of their selected sampling algorithms include at least 66% of the 135 faults. In our study, the t-wise strategies found around 1/3 of the faults found. Anyway, it is hard to compare these results because the set of systems, strategies used as well as the programming language are different and they may provide different results (see Chapter 6). In any event, similar to them, we found t-wise strategies with greater “t” have greater coverage. In our work, a 4-wise strategy found about 1/3 of the faults and in their work a 6-wise strategy found all 135 reported faults.

Variability bugs datasets. Few bug datasets were found in the literature [Abal et al., 2018; Palix et al., 2011; Do et al., 2005; Slaby et al., 2013]. However, only Abal et al. [2018] propose a variability bug dataset. Abal et al. [2018] created a variability bugs dataset (named VBDb) occurring in four highly configurable systems: LINUX, APACHE, BUSYBOX, and MARLIN based on a manual investigation of these projects. Their dataset is composed by 98 variability bugs with a detailed data record about each bug. These bugs comprise common types of errors in systems developed in C and cover different types of feature interactions. The main differences from our study are the number of configurable systems and the programming language. While they investigated four configurable systems, we investigated 30 configurable systems. Indeed the systems of their dataset are larger than the systems of our dataset. However, we manually created the test suite for most of the configurable systems and we also have industry-strength systems, such as ARGOUML. Regarding the programming language, they analyze C systems while we evaluate Java-based systems. This is an advantage,

since practitioners may benefit from a different set of supporting tools and researchers may compare these studies as well as better understand how developers introduce variability bugs in configurable systems.

2.5 Final Remarks

This chapter describes the fundamental concepts used in this thesis and discusses related works. We started by introducing developing configurable systems. Following, we presented two main strategies to develop configurable systems (compile and execution-time). Then, we showed a feature interaction overview and introduced a traditional and simple example of a feature interaction problem adapted from WordPress. Next, we present an overview of this testing configurable software systems. In Section 2.4, we presented studies investigating faults, feature interactions, comparing testing strategies for configurable systems, and variability bugs datasets. Furthermore, we discuss the applications of work related to the chapters of this thesis.

After presenting an overview of configurable systems and configurable system testing in the next chapter, we presented some configurable system testing tools and their particularities. Thus, Chapter 3 presents a systematic mapping study that aims to provide an overview of the state-of-the-art testing tools and strategies for configurable systems.

Chapter 3

Systematic Mapping Study

This chapter reports the planning, execution, analysis, and results of the systematic mapping study (SMS) conducted in this thesis. This mapping study explores the existing testing tools for configurable software systems, understands how this works, and applies the literature tools. The main contribution of this chapter is the identification of the state-of-the-art of testing tools for configurable software systems. In addition, we extend and update four previous secondary studies [Meinicke et al., 2014; Machado et al., 2014; Lopez-Herrejon et al., 2015; Varshosaz et al., 2018]. These studies were found through an *ad hoc* review of the literature. We analyze these previous studies to find out the testing tools for configurable systems reported in them.

For this SMS, we propose two research questions as follows. First, *what are the main characteristics of the testing tools for configurable systems?* We investigate tools' characteristics described in the primary studies and use these tools (when available for download). Second, *what are the main strategies used to test configurable systems?* We observed the classifications commonly presented in the primary studies and grouped the tools found following these classifications. As a result, we found a total of 64 primary studies resulting in 60 tools. We analyze the main characteristics of the tools found (e.g., graphical user interface, and free for use). Moreover, we identify four main strategies used to test configurable systems (e.g., *Sound Testing*, and *Automatic Test Case Generation*).

The remainder of this chapter is organized as follows. Section 3.1 presents the planning of this SMS by demonstrating the protocol adopted for the study. Section 3.2 describes the execution according to the protocol. It also shows the selection process of the relevant papers in this SMS. Section 3.3 shows an overview of primary studies.

Section 3.4 presents and discusses the results of the SMS. Section 3.5 shows an evaluation overview of the testing tools found. Section 3.6 presents some implications of our results for researchers and practitioners. Finally, Section 3.7 discusses the threats to validity and Section 3.8 concludes this chapter.

3.1 Planning

In the planning phase, we define the protocol for conducting the SMS. The activities carried out in this phase were: (i) defining the SMS goal, (ii) specifying the research questions; (iii) selecting the databases to search papers; (iv) constructing the search string to be used; and (v) applying the inclusion and exclusion criteria.

Goal: The goal of this study is to identify and analyze tools reported in the literature for testing configurable software systems. We defined this goal due to the wide number of proposed testing tools and the diversification of strategies described in the literature. We believe that software testers would benefit from such summarization and comparison of tools reported in this study.

Research Questions: The research questions (RQ) aim to investigate and understand the state-of-the-art of testing tools for configurable systems. To achieve the goal of this chapter, we establish two general purpose research questions.

RQ₁ - What are the main characteristics of the testing tools for configurable systems?

RQ₂ - What are the main strategies used to testing configurable systems?

RQ₁ aims to check the key features of testing tools for configurable systems. During the execution of the SMS, we found studies that presented different strategies for testing configurable systems. Furthermore, we observed that tools could be classified according to their strategies. Therefore, we propose RQ₂ to find the main strategies used for testing configurable systems.

Electronic Databases: The electronic databases used for the search of the primary studies were: *ACM Digital Library*¹, *IEEE Xplore*², *Engineering Village*³, *Science Direct*⁴, *Scopus*⁵ and *Springer Link*⁶. These digital libraries have been

¹<http://dl.acm.org/>

²<http://ieeexplore.ieee.org/>

³<https://www.engineeringvillage.com>

⁴<http://www.sciencedirect.com/>

⁵<http://scopus.com/>

⁶<http://link.springer.com/>

chosen because they have a large collection of full research papers. They are indexed for published researches from conferences and journals of great importance to the academic community.

Search String: To identify the relevant papers about testing tools for configurable systems, we formulate a search string to find primary studies related to our mapping. Initially, the key words “*tool*”, “*configurable software systems*” and “*software test*” were defined as the main terms of the expression. After piloting search, we define for synonyms of these terms to refine this expression. We also test different search strings for singular of the plural in the terms used for search.

```
("tool") AND ("configurable software systems" OR "software product lines") AND ("software test" OR "software testing")
```

Inclusion and Exclusion Criteria: The inclusion and exclusion criteria allow classifying each study in the mapping study as a candidate to be included to or excluded from the SMS [Kitchenham and Charters, 2007]. As a SMS may involve a large number of studies, we limited the scope of this SMS to select only full papers. The following inclusion and exclusion criteria were defined.

Inclusion Criteria - Papers published in English, published in Computer Science, available in electronic format, published in conferences and journals, and related to the search string terms.

Exclusion Criteria - Shorter than two pages, secondary studies, duplicated studies.

3.2 Execution

We divide the execution of our SMS into two main phases as demonstrated in Figure 3.1. In the first phase, through a review of four secondary studies [Meinicke et al., 2014; Machado et al., 2014; Lopez-Herrejon et al., 2015; Varshosaz et al., 2018], we selected 25 primary studies. The second phase, following the planning described in Section 3.1, consists in applying the search string in the databases to search primary studies. The inclusion and exclusion criteria were then applied to filter the studies. After Phase 1 and Phase 2, we merged the studies found to answer our research questions.

Phase 1: we analyzed were the studies reported in previous secondary studies [Meinicke et al., 2014; Machado et al., 2014; Lopez-Herrejon et al., 2015; Varshosaz et al., 2018]. The 25 primary studies on testing tools for configurable systems that these papers mentioned were included in our SMS.

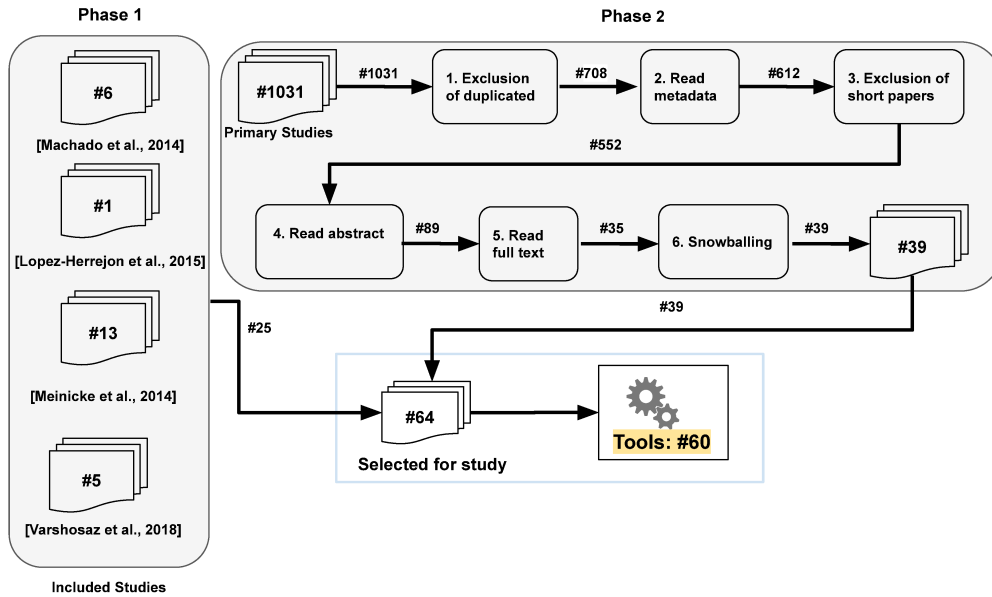


Figure 3.1: Filtering process conducted for study selection

Phase 2: In this phase, our SMS covered studies published between the years 2014 and 2020, since previous studies did not include recent years. To update our initial work that reported works published from 2014 to 2017 [Ferreira et al., 2019], we used the guidelines reported by Wohlin et al. [2020]. We consider all studies returned by databases between the years between 2014 and 2020. The search process was carried out in 2021.

Table 3.1 presents the results regarding the number of papers found in each Electronic Databases. The first column (Studies Returned) presents the scientific databases mined. We obtained 1031 papers after the completion of this process. The third column (Selection) shows the number of remaining papers by electronic database after filtering. We sorted the Table 3.1 data by column third column. *ACM Digital Library* had four works found too in *Springer*, two works found too in *IEEE Xplore*, and one work found too in *Scopus*. Therefore, in total, 39 papers were selected.

Study Selection Process: After completing the step of searching for primary studies, we started a step of filtering these documents. Since we identified a large number of papers in the search phase, the filtering process consisted of six steps. Each step focuses on the inclusion and exclusion criteria and relevance of the study according to its content. Figure 3.1 illustrates each step performed described as follows.

Table 3.1: Studies obtained after the search process.

Data Base	Studies Returned	Selection
ACM Digital Library	110	12
Springer	193	9 (+4)
Engineering Village	74	7
IEEE Xplore	297	5 (+2)
Science Direct	225	4
Scopus	154	2 (+1)
Total	1031	39

Step 1 - Exclusion of duplicates. It involves the elimination of duplicated studies. Studies with same title and authors were discarded. This step eliminated 323 papers, resulting in 708 papers to be analyzed in Step 2.

Step 2 - Read metadata. It consists in removing documents that are not papers. Thus, documents classified as tutorials, posters, panels, lectures, round tables, theses, dissertations, and technical reports were removed in this step. Other studies were discarded because the title did not indicate that the work was about testing configurable systems. This step eliminated 96 papers, resulting in a total of 612 papers to be analyzed in Step 3.

Step 3 - Exclusion of short papers. In this step, we identified the short papers. This step eliminated 60 papers, resulting in a total of 552 papers to be analyzed in Step 4.

Step 4 - Read abstract. The titles and abstracts of the 552 papers selected in Step 3 were checked. Only reading the title and abstract of some papers, it was not possible to have the idea about the content addressed. For this reason, when analyzing some studies, we could not decide about the inclusion or exclusion of them. Therefore, as a way to avoid fast decisions, these studies were included and moved to Step 5 for further reading. This step identified 89 papers with relevance.

Step 5 - Read full text. In this step, we conducted a full-text paper read for 89 papers, aiming to discard papers that does not propose or use testing tools for configurable systems and kept 35 studies.

Step 6 - Snowballing. Searching in electronic databases does not guarantee that all relevant studies on a particular topic would be retrieved. In order to mitigate this limitation, a snowballing process was performed in this step. By applying backward snowballing [Wohlin, 2014] approach, we have included four studies in this step.

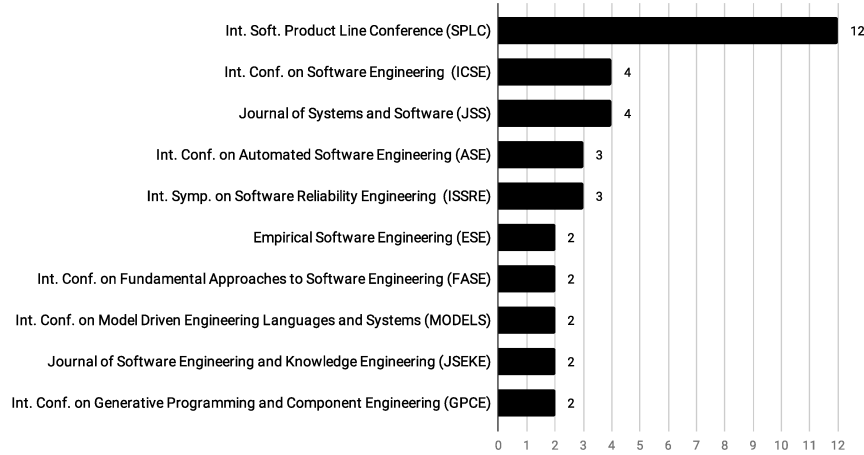
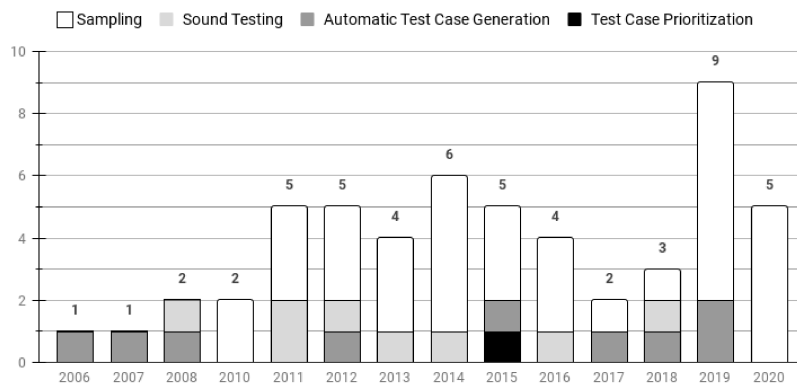


Figure 3.2: Number of testing tools by publication venues

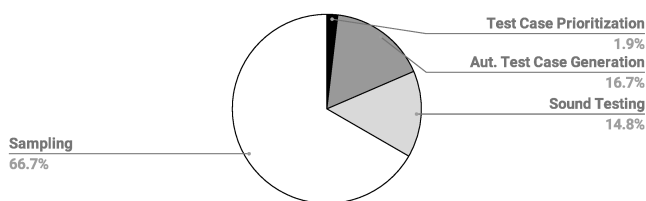
3.3 Overview of Primary Studies

We selected 64 primary studies in total for this mapping study resulting in 60 testing tools for configurable systems. From the primary studies, there were 42 (65.6%) conference papers, 20 (31.3%) journal papers, and two (3.1%) workshop papers. Total number of papers per publication venue is presented in Figure 3.2. We omitted publication venues with only one publication. The publication venues in highlight are the *International Software Product Line Conference (SPLC)*, *International Conference on Software Engineering (ICSE)*, and *Journal of Systems and Software (JSS)*, with, respectively, 12, 4, and 4 papers on testing tools for configurable systems published from 2006 to 2020. Researchers can benefit from this list of publication venues when choosing where to submit their tools.

Figure 3.3 shows the distribution of published tools per year according to the test strategies (Figure 3.3a) and the percentage of tools that implement the test strategies (Figure 3.3b). As Figure 3.3a shows, researchers started to propose testing tools with more intensity after the emergence of tools with the *Sampling* strategy in 2010 (an average of 5 tools per year after 2010, in comparison to an average of 1.3 tools per year before 2010). From 2010 to 2020, *Sampling* was the main strategy used in new tools: 66.7% of the 60 tools found use *Sampling* strategy. The second recurring test strategies found is *Automatic Test Case Generation*, used in 16.7% of the tools found. The strategy with the lowest occurrence is *Test Case Prioritization*, found in only one tool (1.9%) in our primary studies found. Although sampling strategies are more

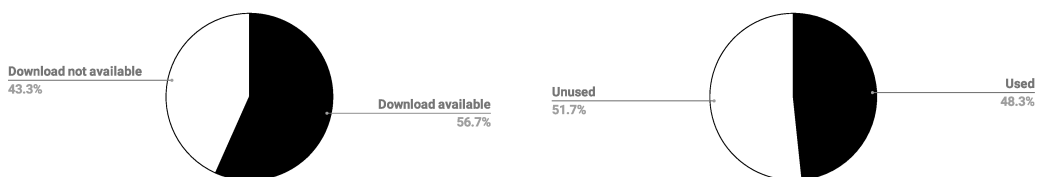


(a) Number of testing tools published by years



(b) Distribution of test strategies

Figure 3.3: Tools distribution over years and testing strategies



(a) Percentage of downloaded tools

(b) Percentage of used tools

Figure 3.4: Distribution of downloaded and tested tools

representative than other strategies, this result demonstrates that the authors of testing tools for configurable systems are still looking for solutions to test configurable systems based on all four major strategies.

Figure 3.4 presents the percentage of tools available for download and the percentage of tools we actually used. As can be seen in the Figure 3.4a, 56.7% of the tools are available for download through their repositories described in their papers. We were able to use about 48% of the tools found (Figure 3.4b). Therefore, not all the tools that were available for download were possible to use. Problems with dependence on the environment to install the tools and problems with input parameters were the main obstacles to the use of these tools.

3.4 Data Analysis

In this subsection, we address each research question presented in Section 3.1. With respect to RQ₁, Table 3.4 presents the 60 testing tools for configurable systems we found in the 64 primary studies. We included testing tools for configurable systems regardless of the strategy implemented by the tool.

***RQ₁** - What are the main characteristics of the testing tools for configurable systems?*

Table 3.4 lists the characteristics of each tool. We defined these characteristics based on previous studies [Pereira et al., 2015; Chen and Babar, 2011; Lee et al., 2012a]. The following characteristics were considered. We indicate whether the tool is accessible for download (Accessible), type of user interaction (TUI), prototype, plugin, free for use (Free), open-source, user guide available (USG), solution examples available (SE). Furthermore, we indicate if we were able to use the tool (Used), code level test (CLT), feature model check (FMC), feature interaction (FI), input data, output data, tool dependency, and tool website. We use "n/a" for information not available.

Regarding the format of the tools, we identified the type of user interaction as command line (LC), graphical user interface (GUI), and online (ONL). We found a total of 34 tools with some type of user interface. The most recurring type of user interface is graphical user interface (30%), followed by command line (28.3%), and online (10%). Additionally, 17 (28.3%) of the tools are described as prototypes, and 20 (33.3%) are plugins for Eclipse. The majority of the tools (56.6%) are described as free for use, and 43.3% are open-source projects.

With respect to the documentation of the tools, we identified that 43.3% tools have user guides available (USG), and 65% have solution examples available (SE). However, we were able to use only 48.3% of the tools. An initial challenge was that only 53.7% of the tools have valid URLs. In addition, some tools have errors of execution or lack of documentation to understand all the requirements for the tools to work correctly.

We found that 63.3% of the tools identify configurations for testing from feature models and 38% use the system's source code. Feature interaction is a situation in which the composition of several features leads to emergent behavior that does not occur when one of them is absent [Svahnberg et al., 2005]. There is a significant percentage of tools aimed at testing the interaction of features. We observed 63.3% of the tools look for feature interactions.

FeatureIDE, Eclipse, and SPLCAT are the most recurring tool dependency found (8.3%, 6.7%, and 3.3%, respectively). XML is the main format for input data (15.6%), followed by DIMACS (8.3%) and GUIDSL (5%). Other formats include SPLX, Clafer, Z3, SXFM, and SPLOT files. Test tools for systems configurable with the aid of a SAT Solver, which takes as inputs such a feature model and a configuration set also represented as a boolean expression. Finally, most of the tools produce outputs in the format of configuration lists (20%) or logs (18.3%). Other output formats are interaction graphs (used in VarXporer [VarXplorer, 2018]) and JUnit reports (used in ParTeG [ParTeg, 2008]). List Configurations are usually represented as CSV, TXT, and XML files.

Table 3.2: Characteristics each tool

Tool [Ref.]	Accessible	TUI	Prototype	Plugin	Free	Open-source	USG	SE	Used	CLT	FMC	FI	Input date	Tool Dependency	Output date	Website
Baital [Baranov et al., 2020]	•	LC		•	•	•	•	•	•	•	•	•	DIMACS		CL	[Baital, 2020]
YASA [Krieter et al., 2020]	•	GUI		•	•	•	•	•	•	•	•	•	XML	FeatureIDE	CL; LOG	[YASA, 2020]
VarXplorer [Rocha et al., 2020]	•	GUI		•	•	•	•	•	•	•	•	•		VarexJ; Varviz	Interaction Graph	[VarXplorer, 2018]
Nautilus/VTSP [Ferreira et al., 2020e]	•	ONL		•	•	•	•	•	•	•	•	•		Nautilus Framework		[Nautilus-VTSP, 2018]
Hasan [Hasan et al., 2020]	•	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	•	•	n/a	n/a	n/a	n/a	n/a
GrES [Hierons et al., 2020]	•	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	•	•	n/a	n/a	n/a	n/a	[GrES, 2019]
CoPTA [Luthmann et al., 2019b]	•	LC		•	•	•	•	•	•	•	•	•				[CoPTA, 2019]
Kaltenecker [Kaltenecker et al., 2019]	•	LC		•	•	•	•	•	•	•	•	•	XML	SPLConq.	LOG	[Sampling, 2019]
Smarch+BBPF [Munoz et al., 2019]	•	LC		•	•	•	•	•	•	•	•	•	Clafex; DIMACS; Z3			[Smarch+BBPF, 2019]
Luthmann [Luthmann et al., 2019a]	•	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	•	•	n/a	n/a	CoPTA		
Al-Hajjaji [Al-Hajjaji et al., 2016c, 2014]	•	GUI		•	•	•	•	•	•	•	•	•	XML	FeatureIDE	CL; LOG	[Luthmann, 2019]
ECCO [Fischer et al., 2019]	•	n/a	n/a							•	•	n/a	n/a	n/a	n/a	n/a
MOCSSFO [Ramgouda and Chandraprakash, 2019]	•			•	•	•	•	n/a	•	•	•	•		n/a	n/a	n/a
ConFTGen [Fragal et al., 2019]	•	GUI	•	•	•	•	•	n/a	•	•	•	•	n/a	Eclipse; FeatureIDE	n/a	[ConFTGen, 2019]
CoPRO [Nguyen et al., 2019]	•	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	•	•	•	n/a	n/a	n/a	n/a
Yan [Yan et al., 2019]	•	n/a	n/a	n/a	n/a	n/a	n/a	•	•	n/a	•	n/a	XML	n/a	n/a	n/a
Arrieta [Arrieta et al., 2019]	•	n/a	n/a	n/a	n/a	n/a	n/a	•	•	•	•	n/a	n/a	FeatureIDE	n/a	[Arrieta, 2019]
VarexC [Wong et al., 2018]	•	LC		•	•	•	•	•	•	•	•	•			LOG	[VarexC, 2018]
Ruland [Ruland et al., 2018]	•	LC		•	•	•	•	•	•	•	•	•		CPAchecker; CPATiger; ICPL; SiMPOSE	n/a	[Ruland, 2018]
Li [Li et al., 2018]	•	n/a	n/a	n/a	n/a	n/a	n/a	•	•	•	n/a	n/a	n/a	n/a	n/a	n/a
Filho [L. et al., 2018]	•	n/a	n/a	n/a	n/a	n/a	n/a	•	•	n/a	•	n/a	n/a	JMetal	n/a	n/a
FMTS [Ferreira et al., 2017]	•	n/a	•	•	•	n/a	n/a	n/a	•	•	•	•	n/a	FaMA	n/a	n/a
S-SPLat [Souto et al., 2017]	•	LC		•	•	•	•	•	•	•	•	•	GUIDSL		LOG	[REFRACT, 2014]
Al-Hajjaji [Al-Hajjaji et al., 2016b]	•	GUI	•	•	•	•	•	•	•	•	•	•	XML	n/a	CL; LOG	[Al-Hajjaji, 2016]
InclLing [Al-Hajjaji et al., 2016a]	•	GUI; LC	n/a	•	•	•	n/a	•	•	•	•	•	XML	FeatureIDE	CL; LOG	[InclLing, 2016]
TESALIA [Galindo et al., 2016]	•	GUI; ONL	n/a	•	•	•	n/a	•	•	•	•	•	n/a	n/a	n/a	[TESALIA, 2016]
VarexJ [Meinicke et al., 2016]	•	LC		•	•	•	•	•	•	•	•	•	DIMACS	log	n/a	[VarexJ, 2016]
Matnei-Filho [Matnei Filho and Vergilio, 2016]	•	n/a	n/a	n/a	n/a	n/a	n/a	•	•	•	•	n/a	XML	FMTS	n/a	n/a
CPA/tiger [Bürdek et al., 2015]	•	LC		•	•	n/a	•	•	•	•	•	•		CPAchecker		[CPA/TIGER, 2015]
CTWeb [Lamancha et al., 2015]	•	ONL	•	n/a	•	n/a	n/a	•	•	•	•	•	n/a	n/a	n/a	n/a
TEMSA [Wang et al., 2015]	•	ONL		•	•	•	n/a	•	•	•	•	•	n/a	n/a	n/a	[TEMSA, 2015]
Reuling [Reuling et al., 2015]	•	GUI		•	•	•	•	•	•	•	•	•	SXFM; SPLX	Eclipse; SPLCAT	n/a	[Reuling, 2019]
Arcaini [Arcaini et al., 2015]	•	n/a	n/a	n/a	n/a	n/a	n/a	•	•	•	•	•	XML		n/a	n/a
Henard [Henard et al., 2013b]	•	GUI		•	•	•	•	•	•	•	•	•	DIMACS; XML	PLEDGE	CL	[Johansen, 2012]
Sánchez [Sánchez et al., 2014]	•	LC		•	•	•	•	•	•	•	•	•	XML	SPLAR; BeTTY; SPLCAT	n/a	[Sánchez, 2012]
FeatureIDE [Thüm et al., 2014]	•	GUI		•	•	•	•	•	•	•	•	•	GUIDSL; XML; SPLConq.	Eclipse	CL; LOG	[FeatureIDE, 2014]
MPLM [Samih and Bogusch, 2014]	•	GUI		•	•	n/a	n/a	•	•	•	•	•	pure::variants	Eclipse	n/a	n/a
PLEDGE [Henard et al., 2013a, 2014a]	•	GUI		•	•	•	•	•	•	•	•	•	DIMACS		CL	[PLEDGE, 2014]
REFRACT [Swanson et al., 2014]	•	ONL	•	n/a	•	n/a	n/a	n/a	•	•	•	•	n/a	Rainbow	n/a	[REFRACT, 2014]
Varex [Nguyen et al., 2014]	•	GUI; ONL		•	•	•	•	•	n/a	n/a	•	•	n/a	n/a	n/a	n/a
Marijan [Marijan et al., 2013]	•	n/a	n/a	n/a	n/a	n/a	n/a	n/a	•	•	•	•	XML	PACOGEN	n/a	n/a
IPT [Wang et al., 2013]	•	n/a	n/a	•	n/a	n/a	n/a	n/a	•	•	•	•	n/a	n/a	n/a	n/a
SPLAT [Kim et al., 2013]	•	LC		•	•	•	•	•	•	•	•	•	GUIDSL		LOG	[SPLAT, 2013]
Henard [Henard et al., 2014b]	•	LC	n/a	•	•	•	•	•	•	•	•	•	DIMACS		n/a	[Henard, 2020]
Shi [Shi et al., 2012]	•	n/a	n/a	n/a	n/a	n/a	n/a	•	•	•	•	•	n/a	n/a	n/a	n/a
ICPL [Johansen et al., 2012a]	•	GUI; LC	n/a	•	•	•	n/a	•	•	•	•	•	XML	FeatureIDE	CL; LOG	[ICPL, 2012]
MATE [Seiger and Schlegel, 2012]	•	n/a	•	•	n/a	n/a	n/a	n/a	•	n/a	n/a	n/a	n/a	Eclipse	n/a	n/a
shared-execution [Kim et al., 2012b]	•	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	•	•	•	n/a	n/a	n/a	n/a
Chvatal [Johansen et al., 2012b]	•	n/a		•	•	•	•	•	•	•	•	•	n/a	n/a	n/a	n/a
CASA [Garvin et al., 2011]	•	GUI; LC		•	•	n/a	•	•	•	•	•	•	XML	FeatureIDE	CL; LOG	[CASA, 2014]
MoSo-PoLiTe [Oster et al., 2011, 2010]	•	n/a	n/a	•	n/a	n/a	n/a	n/a	•	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Pacogen [Hervieu et al., 2011]	•	LC		•	•	•	•	•	•	•	•	•	SPLIT	Eclipse	CL	[Pacogen, 2011]
SPLCAtool [Johansen et al., 2011]	•	GUI		•	•	•	•	•	•	•	•	•	GUIDSL; DIMACS		CL	[SPLtool, 2013]
SPLTester [Kim et al., 2011a]	•			•	n/a	n/a	n/a	•	•	•	•	•		Eclipse		
SPLverifier [Apel et al., 2013c, 2011]	•	LC		•	•	n/a	n/a	n/a	•	•	•	•		JavaPathfinder		[SPLVerifier, 2011]
Kesit [Uzuncaova et al., 2008]	•	n/a	•	n/a	n/a	n/a	n/a	n/a	•	n/a	n/a	•	n/a	n/a	n/a	n/a
SPLMonitor [Kim et al., 2010]	•			•	•	•	•	•	•	•	•	•				
ParTeG [Weißleder et al., 2008]	•	GUI		•	•	n/a	n/a	•	•	•	•	•		Eclipse	JUnit report	[ParTeg, 2008]
GATE [Feng et al., 2007]	•			•	n/a	n/a	n/a	n/a	•	•	•	•				
Asadal [Kim et al., 2006]	•	GUI		•	•	n/a	n/a	n/a	•	•	•	n/a	n/a	n/a	n/a	n/a
Total	34		17	20	34	26	26	39	29	23	38	38				

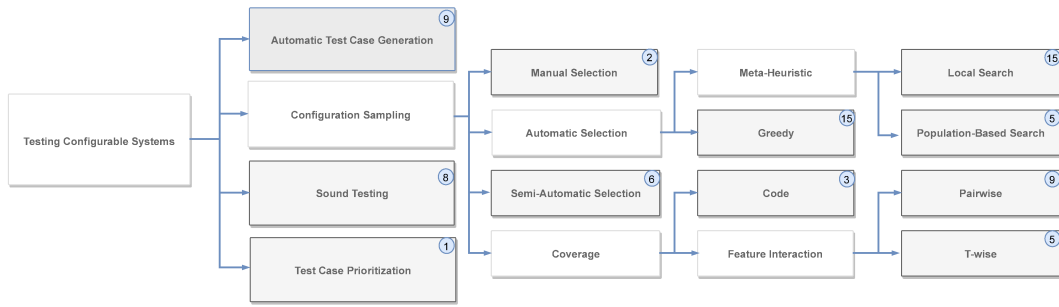


Figure 3.5: Distribution of testing strategy tools

RQ₂ - *What are the main strategies used to testing configurable systems?*

In order to classify the tools in relation to their testing strategy, we used the classification scheme proposed by Varshosaz et al. [2018]. Figure 3.5 shows the number of tools found that implement each strategy, according to the classification of Varshosaz et al. [2018]. A brief description of each testing strategy is presented in Table 3.3.

The *Configuration Sampling* strategy is the most recurring in our sample. However, it is further classified in subcategories that, in turn, may have their own subcategories. For instance, the *Configuration Sampling* strategy comprises the strategies *Manual Selection*, *Automatic Selection*, *Semi-automatic Selection*, and *Coverage*. The *Automatic Selection* strategy is the most recurring, with 15 occurrences of *Greedy* strategy, 15 occurrences of *Local Search* strategy, and 5 occurrences of *Population-Based Search* strategy. Seventeen tools implement *Coverage* strategy, being the *Code* (3), *Pairwise* (9), and *T-wise* (5). Six tools implement *Semi-automatic Selection* and two tools implement *Manual Selection*.

In addition to the *Configuration Sampling* strategy, we found 9 tools implementing *Automatic Test Case Generation* strategy, and 8 tools implementing *Sound Testing* strategy. There was only a single occurrence of *Test Case Prioritization* strategy. As can be seen in Figure 3.5 the total number of occurrences is greater than the number of tools found because a tool can implement more than one test strategy. This classification might be useful to guide researchers in describing their test strategies and to identify strategies that need further investigation.

Table 3.4 shows which testing strategies each tool implements as shown in Table 3.3. Table 3.4 highlighted in the main test strategies. We identified some testing strategies according to the works that describe each testing tool. Thus, we have collected a total of 14.8% tools that use the *Sound Testing* strategy. For the strategies

Table 3.3: Testing strategies description

Strategy	Description
Automatic Test Case Generation	Strategies are based on heuristic techniques for determining appropriate test cases for testing configurable systems
Configuration Sampling	Uses techniques to test only a representative subset of all possible configuration
Sound Testing	Use to test all possible configurations of the configurable systems
Test case prioritization	Orders test cases for execution in an order that attempts to increase their effectiveness at meeting some performance goals
Manual Selection	Configurations are selected for testing manually
Semi-Automatic Selection	Test configurations are selected based on some parameter for selection such as coverage on feature interactions
Automatic Selection	Configurations are selected according to some strategy of prioritizing configurations for testing
Meta-Heuristic	It is strategies aim at selecting a subset of configuration as an optimal solution for this problem using computational search in a configuration space
Greedy	The approach selects the best local solution to select the most representative configurations for testing according to the criteria adopted in each approach
Local Search	Such approaches start with an initial set of configurations and each iteration seeks to extend the set of selected configurations
Population-Based Search	The initial approach with initial set of configurations that mutated and recombined into new configurations. A fitness function is usually used as a measure for evolving the set of solutions during the process
Coverage	It is a criterion used to guarantee the quality of the qualifications selected for testing. For example, feature can be used interaction coverage
Code	Generally, approaches list a metric to measure the configurable system source code and check the coverage of the test configurations against the metric used
Feature Interaction	The approaches evaluate sample configurations by checking feature interaction. T-wise can be used to check the coverage of feature interaction
T-wise	Select a subset of configurations that covers a valid group of t features being activated and deactivated simultaneously (2-wise, also known as pairwise)

sampling found *Manual Selection*, *Automatic Selection*, *Semi Automatic Selection*, and *Coverage* found 3.3%, 15%, 11.7%, and 28.3% , respectively. We also identified a small number of tools of which the strategy for configurable systems is *Test Case Prioritization*. The total for this strategy is 1.7% of the overall tools found. Finally, we identified 15% for *Automatic Test Case Generation*.

Table 3.4: Testing strategies

Tool [Ref.]	Sound Testing	Manual Selection	Automatic Selection	Semi Automatic Selection	Coverage	Test Case Prioritization	Automatic Test Case Generation
Baital [Baranov et al., 2020]			Greedy		T-wise		
YASA [Krieter et al., 2020]			Greedy		T-wise		
VarXplorer [Rocha et al., 2020]				•	Pairwise		
Nautilus/VTSP [Ferreira et al., 2020e]			Local search				
Hasan [Hasan et al., 2020]			Local search				
GRES [Hierons et al., 2020]					Pairwise		
CoPTA [Luthmann et al., 2019b]			Greedy				•
Kaltenecker [Kaltenecker et al., 2019]			Population-based search				
Smarch+BBPF [Munoz et al., 2019]			Local search				
Luthmann [Luthmann et al., 2019a]			Greedy				
Al-Hajjaji [Al-Hajjaji et al., 2016c, 2014]			Local search				
ECCO [Fischer et al., 2019]							•
MOCSFO [Ramgouda and Chandraprakash, 2019]			Local search				
ConFTGen [Fragal et al., 2019]							
CoPRO [Nguyen et al., 2019]							
Yan [Yan et al., 2019]			Local search				
Arrieta [Arrieta et al., 2019]			Local search				
VarexC [Wong et al., 2018]	•						
Ruland [Ruland et al., 2018]			Population-based search		Pairwise		
Li [Li et al., 2018]							•
Filho [L. et al., 2018]			Local search		Pairwise		
FMTS [Ferreira et al., 2017]							•
S-SPLat [Souto et al., 2017]			Greedy		Pairwise		
Al-Hajjaji [Al-Hajjaji et al., 2016b]		•					
InCLing [Al-Hajjaji et al., 2016a]			Greedy		Pairwise		
TESALIA [Galindo et al., 2016]			Population-based search				
VarexJ [Meinicke et al., 2016]	•						
Matnei-Filho [Matnei Filho and Vergilio, 2016]			Population-based search		Pairwise		
CPA/tiger [Bürdek et al., 2015]							•
CTWeb [Lamancha et al., 2015]			Greedy		Pairwise		
TEMSA [Wang et al., 2015]						•	
Reuling [Reuling et al., 2015]			Greedy				
Arcaini [Arcaini et al., 2015]			Greedy				
Henard [Henard et al., 2013b]			Local search				
Sánchez [Sánchez et al., 2014]			Local search				
FeatureIDE [Thüm et al., 2014]		•					
MPLM [Samih and Bogusch, 2014]				•			
PLEEDGE [Henard et al., 2013a, 2014a]			Local search				
REFRACT [Swanson et al., 2014]				•			
Varex [Nguyen et al., 2014]	•						
Marijan [Marijan et al., 2013]			Local search				
IPT [Wang et al., 2013]			Local search				
SPLAT [Kim et al., 2013]	•						
Henard [Henard et al., 2014b]			Population-based search	•			
Shi [Shi et al., 2012]			Greedy				
ICPL [Johansen et al., 2012a]			Greedy		T-wise		
MATE [Seiger and Schlegel, 2012]							•
shared-execution [Kim et al., 2012b]	•						
Chvatal [Johansen et al., 2012b]			Greedy	•			
CASA [Garvin et al., 2011]			Local search		T-wise		
MoSo-PoLiTe [Oster et al., 2011, 2010]			Greedy	•	Pairwise		
Pacogen [Hervieu et al., 2011]			Local search				
SPLCAtool [Johansen et al., 2011]			Greedy		T-wise		
SPLTester [Kim et al., 2011a]	•						
SPLVerifier [Apel et al., 2013c, 2011]	•						
Kesit [Uzuncaova et al., 2008]	•						
SPLMonitor [Kim et al., 2010]			Greedy		Code coverage		
ParTeG [Weißleder et al., 2008]							•
GATE [Feng et al., 2007]							•
Asadal [Kim et al., 2006]							•
Total	8	2	Greedy: 15 Local search: 15 Population-Based Search: 5	6	Pairwise: 9 T-wise: 5 Code: 3	1	9

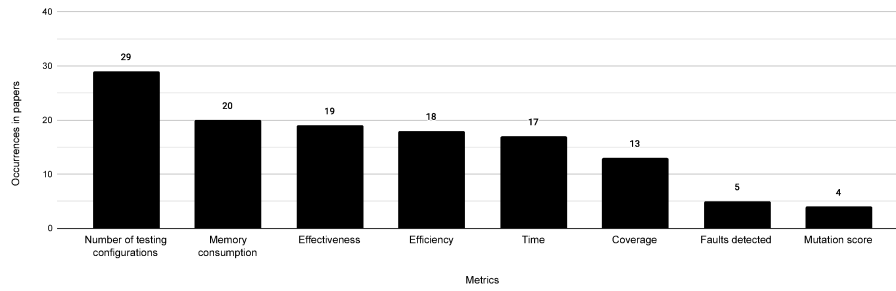
3.5 Evaluation Overview of Testing Tools for Configurable Systems

This section discusses the strategies used in the primary studies to evaluate the testing tools. We present an evaluation overview of testing tools for configurable systems described in the papers found in this mapping study. We found that about 87% of the papers present some evaluation of the tools reported and most present as evaluation metrics, target configurable systems, and approaches to comparison. Metrics are usually related to the research questions reported by the papers. Target configurable systems are used to observe the behavior of the proposed tools. In addition, the papers use other approaches consolidated in the literature to serve as a comparison. These approaches can be algorithms, methods, and other testing tools.

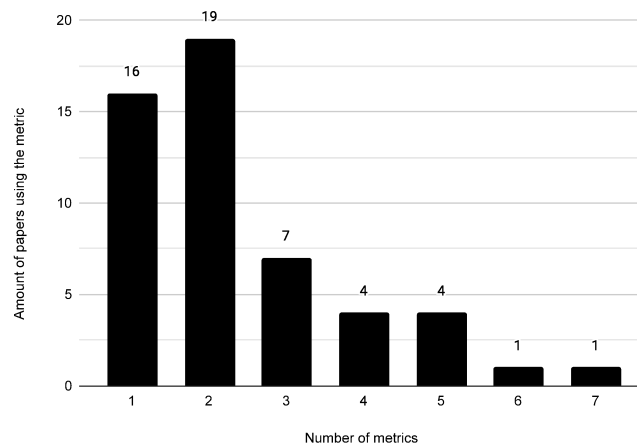
Evaluation metrics. Figure 3.6a shows the metrics used to evaluate configurable system testing tools. We only present metrics with four or more occurrences. As an example, for the metric *Number of test configurations* found 29 occurrences, followed by *Memory consumption* found 20 occurrences. We counted how many metrics the papers use to evaluate the proposed configurable system testing tools, Figure 3.6b presents this summary. As can be seen in Figure 3.6b it shows that 19 papers use two metrics for evaluation and only one paper uses seven metrics for evaluation. As can be seen, the papers use few metrics for evaluating configurable system test tools. Researchers wishing to do an assessment of their tools can look at these results to guide them in designing their assessments.

Evaluation configurable systems. Table 3.5 presents a summary of the main systems used to evaluate test strategies for configurable systems. As can be seen, the systems ELECTRONIC SHOPPING and GPL were the most used in evaluations of test tools, and both appear in the validation of 14 tools. We have omitted the systems that were used in only one paper. We note that the papers use few systems to evaluate the tools. 30 papers use up to five target systems, 18 papers use between six and ten target systems, only two papers use more than ten target systems as evaluation.

Evaluation tools. Figure 3.7 presents a summary of the main approaches that the papers use to evaluate the proposed tools. As can be seen the *NSGA-II*, *Random* and *CASA* where the highest approaches, mentioned in 8, 8 and 6 papers, respectively. We have omitted the approaches that were used as an assessment in just one paper. We found that 14 papers use up to two approaches as comparison, we also found 14 papers that use 3 to 4 approaches as comparison. Finally, we found 2 papers that use 5 to 6 approaches as comparison of their proposed tools.



(a) Number of metrics



(b) Paper distribution by metrics

Figure 3.6: Evaluation metrics found

Table 3.5: Evaluation configurable systems found

Name	Total	Name	Total	Name	Total
Electronic Shopping	14	GPL	14	Smart Home	11
Berkeley DB	10	E-Mail	9	DesktopSearcher	6
Web Portal	6	Apache	5	Coche Ecologico	5
Counter Strike	5	Elevator	5	FreeBSD	5
Mine-Pump	5	Arcade Game	4	Bugzilla	4
BusyBox	4	CAS	4	eCos	4
GCC	4	James	4	Linux kernel	4
Prevayler	4	Video Player	4	AVG	3
Cellphone	3	Dell	3	Electronic Drum	3
JTopas	3	Notepad	3	Printers	3
SPL SimulES	3	Violet	3	ZipMe	3
Companies	2	axTLS	2	BankingSoftware	2
BattleofTanks	2	Car Software System	2	Checkstyle	2
CISCO	2	Doc Generation	2	Drupal	2
Dune	2	ETGC	2	Fiasco	2
FM Test	2	GC	2	HiPAcc	2
Inventory	2	Jetty	2	LLVM	2
QuEval	2	Sienna	2	Spin-S	2
Spin-V	2	Sudoku	2	uClibc-ng	2
X264	2	XStream	2	Fame DBMS	2

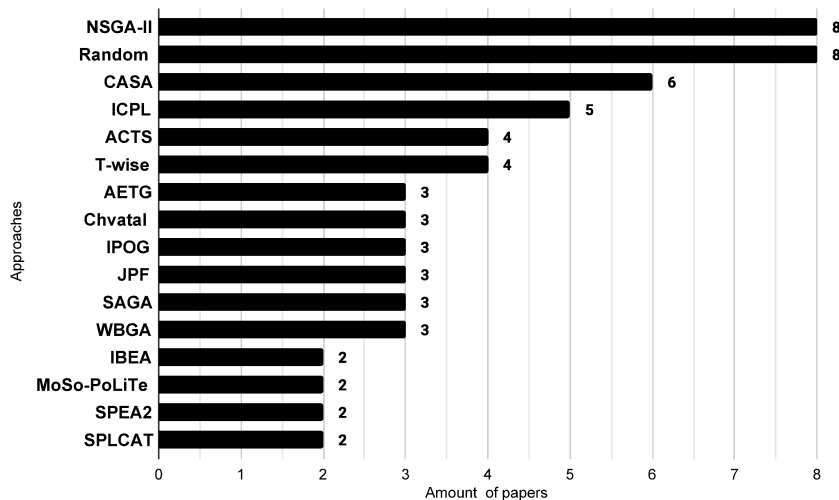


Figure 3.7: Evaluation approaches found

3.6 Implications for Researchers and Practitioners

Researchers may be benefiting from this mapping study in several aspects. First, researchers can look at the tools' characteristics and propose new tools to cover features with few representation. Furthermore, based on the raised characteristics, the researchers may have a guide to describe their tools. Second, it is possible to observe what are the testing strategies implemented by the tools. Third, it is possible to observe how to evaluate their tools and the main events the researchers publish their tools.

Through our mapping study, it is possible to observe that the tools found in the majority are sampling strategies and that strategies to generate configurations for testing look more often only for the feature model of the configurable systems. Furthermore, few strategies observe the characteristics of the source code of the configurable systems to generate configurations for testing. Researchers can cover this lack in the literature by proposing strategies that can generate priority configurations for testing through the source code of the configurable systems, for example, observing the features with more predominance in the source code and identifying the other features that interact with the most prevalent ones.

In this map, 64 tools were found and proposed from 2006 to 2020 years. However, an effective and efficient tool that stands out among the others unknown. Therefore, it is necessary to study the comparison of the tools. We invite researchers to observe our results and compare the available tools through empirical studies. We list the characteristics reported in the papers of the tools found. Also, through Section 3.5 it is

possible to observe an Overview on the evaluation of the proposed tools. Researchers can be guided by our opinions and design future works to evaluate and propose test tools for configurable systems.

Testing activity for configurable systems is very costly. Our work contributes to practitioners' need to have the necessary tooling support to learn about new tools through their main characteristics and strategies. Practitioners can benefit from our mapping study and have an initial guide to discovering new tools, saving time and effort to find tools that best suit their preferences and constraints. Aiming at supporting the choice of a testing strategy, we condensed our results in Tables 3.4 and Tables 3.4 representing the characteristics and strategies of the tools separately. Testers can extend their set of user testing tools or incorporate new testing strategies into testing activity. With the 60 test tools presented, testers can expand testing configurable systems, either with a more current tool to replace a used strategy or even incorporate new test strategies.

3.7 Threats to Validity

A key issue in our SMS is the validity of the results. Even with careful planning, different factors may affect these research results. This section discusses decisions we made to reduce the impact of these factors on the study validity. These actions are mainly related to the research method we adopted to increase the study confidence.

Review Scope and Search Strategy. We selected six different electronic databases for the SMS, but there might be relevant papers in other databases. To minimize this threat, we rely on the use of databases that aggregate papers from diversified publishers, such as Scopus and Engineering Village. With respect to the search strategy, we designed a search string for filtering results. This string includes the most common terms for “configurable software systems” and “testing tools”. We expected achieved a sufficient number of relevant papers in the studied context. In addition, we performed a pilot search to define the terms to appear in the final search string. However, we cannot assume that all existing related papers were found by this filtering strategy.

Manual Filtering. To eliminate repeated papers and keep only relevant work, one researcher verified the inclusion and exclusion criteria manually and another researcher audited the results. In this filtering process, we performed three steps. First, the Ph.D. candidate read each metadata paper and classified it as “out of scope”, “unsure for revision”, “papers that use a tool”, and “papers that propose a tool”. We discarded all papers classified as out of scope. In the second step, another researcher repeated the

previous procedure. Finally, a third researcher decided to include or not a paper based on the classification given by the other two researchers. In case of conflicting classification, the paper was included for the next step. We believe this protocol minimized biases by considering the point of view of three researcher, and the agreement of at least two of them.

Full-text Analysis and Data Extraction. The Ph.D. candidate was responsible to read fully the selected papers and extract information about the testing tools. In case of doubts, he discussed with at least one other researcher. Although the careful conduction of this, with no deadline for completion, we do not take other measures for risk reduction. In this context, some tools may have been wrongly discarded. For instance, we discarded tools, such as CPAChecker [Dirk and Keremoglu, 2011], because we considered their focus not related to testing of configurable systems.

Identifying Features and Comparing Tools. We were not able to find the name and characteristics of some tools. In these cases, we named a tool with the authors who proposed it and tried our best to infer some missing information. Some tools were, in fact, an evolution of previous tools and, in such cases, we kept the name of the previous tool.

3.8 Final Remarks

This chapter presented a SMS to identify the state-of-the-art testing of tools for configurable systems. The goal was to explore the existing testing tools for configurable systems and understand how it works. We mined six scientific databases (ACM Digital Library, IEEE Xplore, Engineering Village, Science Direct, Scopus, and Springer) and retrieved 64 primary studies. In this primary studies we found a total of 60 testing tools for configurable systems and classified them according to 16 characteristics and four main testing strategies.

The SMS presented in this chapter can benefit developers and testers to find configurable tools that best fit their needs. Furthermore, researchers can benefit from the study presented in this chapter, because in addition to the list of relevant papers that describe the tools, as the testing strategies used. We analyzed the tools found concerning the 16 characteristics and four main testing strategies. This thesis shows an overview of 64 primary studies found and presents an overview of how the researchers evaluated testing tools for configurable systems found. Finally, we discuss our results regarding implications for researchers and practitioners. We have listed all papers used in the proposed mapping study steps [Ferreira et al., 2020a].

The next chapter describes the process of creating a test-enriched dataset for configurable systems. Our dataset contains some of the most configurable systems reported in the literature and known repositories. To the best of the PhD candidate knowledge, this dataset is the first one for configurable systems with an extensive test suite [Ferreira et al., 2020b].

Chapter 4

A Test-enriched Dataset for Configurable Software Systems

Despite the number of studies on testing configurable software systems, previous work [Engström and Runeson, 2011; Machado et al., 2014; Lopez-Herrejon et al., 2015] report a lack of empirical evaluations, including a community-wide dataset, to guide the comparison of different testing approaches, strategies, and methods. To fill this gap, we proposed a test-enriched dataset of configurable systems [Ferreira et al., 2020b]. Although we have found 60 configurable systems developed in Java-based programming languages, only ten configurable systems have their available test suite. Aiming at increasing the number and scope of systems in our dataset, we created a test suite for further 20 systems. Based on previous studies, the creation of tests followed two reasonable thresholds to increase the testing suite’s quality: 70% of code coverage [Offutt et al., 2003] and 40% of killed mutants [Aaltonen et al., 2010] for all 20 systems of the dataset. These two criteria aim at providing a high-quality testing dataset. At the end, we created a dataset with 30 test-enriched configurable systems.

This chapter presents the five steps on the dataset creation. Figure 4.1 presents an overview of these five steps. We describe each part of Figure 4.1 in Section 4.1 to Section 4.4. First, we report how we selected subject projects (Section 4.1). Second, we describe how we translate systems developed in other variability techniques into variability encoding (Section 4.2). Third, we report the test suite creation (Section 4.3). Fourth, we describe the metric collection process (Section 4.4). Finally, we provide an overview of the test-enriched dataset (Section 4.5). We also discuss the threats to the validity of the study (Section 4.6) and concludes this chapter (Section 4.7).

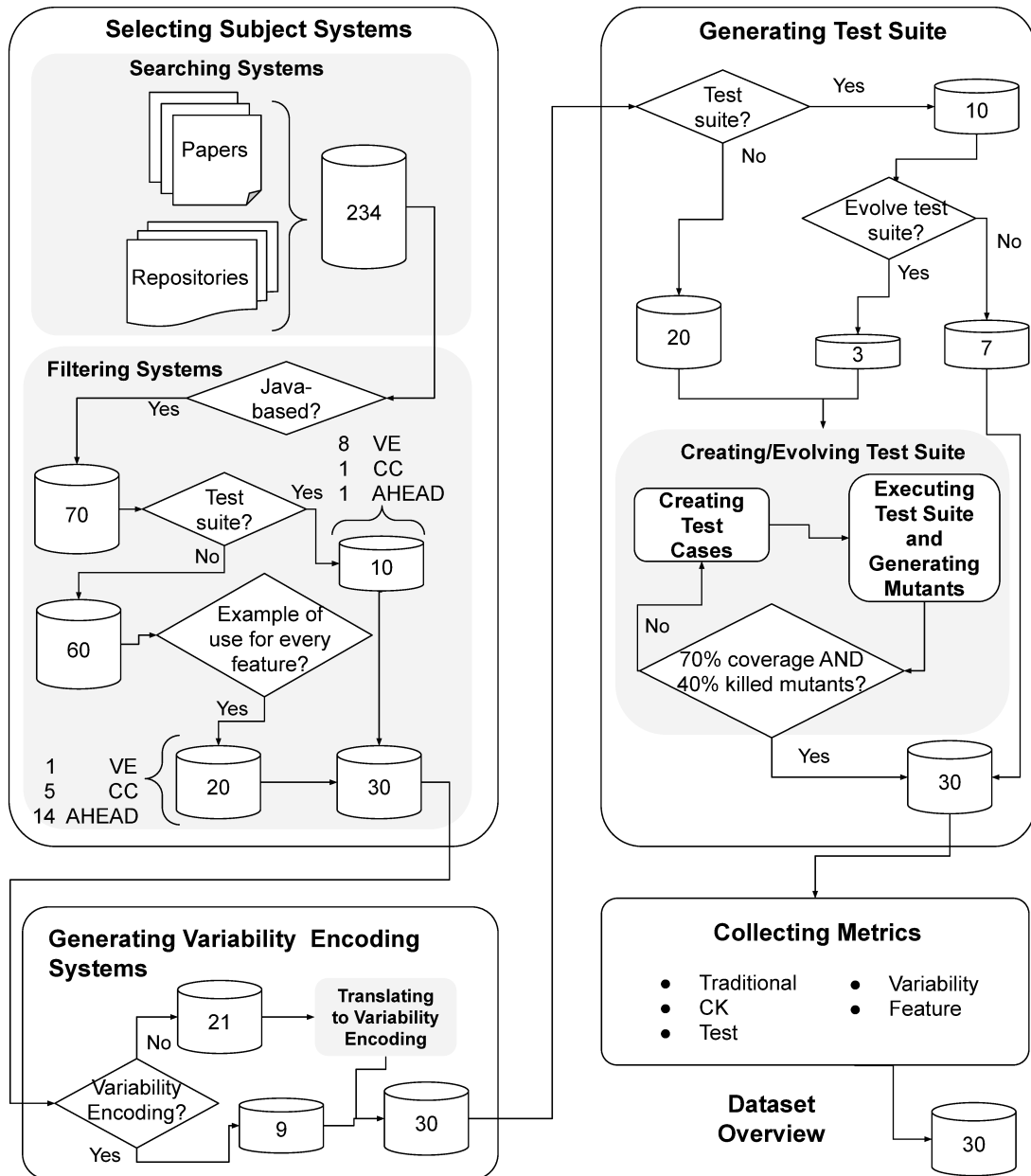


Figure 4.1: Dataset creation process

4.1 Selecting Subject Systems

As shown in Figure 4.1, we searched for configurable systems on six survey papers about testing configurable systems [da Mota et al., 2011; Engström and Runeson, 2011; Lee et al., 2012b; Lamancha et al., 2013; Machado et al., 2014; Lopez-Herrejon et al., 2015] and on all primary studies found on them. In addition, we included configurable systems of three well-known repositories of configurable systems: SPL2GO [SPL2go, 2020], SPL REPOSITORY [Vale et al., 2015b], and ESPLA CATALOG [Martinez et al., 2017]. At the end of this initial search, we found 234 configurable systems. Then, given tool constraints, we limited our dataset to configurable systems developed in a Java-based programming language (i.e., in JAVA with variability encoding, conditional compilation, or AHEAD). As a result, only 70 configurable systems remain in our dataset. By analyzing these 70 systems, we noted that only 10 of them have a test suite available (8 developed with variability encoding, 1 with conditional compilation, and 1 with AHEAD). Consequently, only these 10 systems would compose our dataset. Aiming at increasing the number of systems in our dataset, we looked at the remaining

60 configurable systems developed in a Java-based programming language without a test suite. Considering that we need a deep understanding of the systems as well as their features to develop a test suite, we looked at each system’s documentation searching for information. Once we found an example of use for each available feature of a target configurable system, we included it into our dataset. As an outcome of this analysis, we selected 20 additional configurable systems (i.e., 1 with variability encoding, 5 with conditional compilation, and 14 in AHEAD). At the end, we selected 30 configurable systems to compose our dataset.

4.2 Generating Variability Encoding Systems

Our dataset is composed of annotated systems with variability through variability encoding (Chapter 2). For the 21 configurable systems not developed using variability encoding (i.e., 15 and 6 written in AHEAD and conditional compilation, respectively), we translated their code to variability encoding, as show in Figure 4.1. Among the strategies to introduce variability in software systems, variability encoding has drawn practitioners’ attention since developers only need to *annotate variation points* on their existing systems. Therefore, developers simply activate or deactivate features to

address different deployment contexts. For short, while annotating variation points, developers should create a configuration file where they determine options that are going to be active in a target variation.

Aiming at facilitating the testing suite execution, we translate the (6) conditional compilation and (15) AHEAD systems into variability encoding systems. For conditional compilation systems, we manually converted the compilation directives into variability encoding. As the translation is straight forward (e.g., `#ifdef(FEATURE)` into `if(Configuration.FEATURE)`), chances of manual errors are minimized. For AHEAD systems, we relied on FEATUREIDE to automatically translate AHEAD code into a variability encoding code. FEATUREIDE [Thüm et al., 2014] is an open-source integrated development environment on Eclipse that supports several implementation techniques for feature-oriented software development, such as AHEAD, FEATUREC++, and ASPECTJ. After translation of each system, we extensively, run test cases to identify eventual transformation errors.

4.3 Creating Test Suite

As shown in Figure 4.1, for the 20 systems without a test suite, we created the test suite from scratch. Aiming at enriching discussions and test suite coverage, we extend the test suite of other three systems. The creation/extension of the test suites consist of creating testing cases, executing the test suite with mutants, and checking if the stop criteria are satisfied, as we explain next.

Creating testing cases. We use JUNIT framework [JUnit, 2020], FEST [FEST, 2020], and MOCKITO [Mockito, 2020] for creating testing cases, writing tests for systems with graphical user interfaces, and creating mock objects which simplifies the development of tests for classes with external dependencies, respectively.

Executing mutation testing. For each subject system, we generated a single configuration containing as many active features as possible. We then use PIT [Coles et al., 2016] to generate mutants on each configuration and execute the respective test suite against them. Based on the PIT’s execution report, we decided whether it was necessary to create more test cases aiming at increasing killed mutants coverage.

Checking stop criteria fulfillment. We use JACOCo [JaCoCo, 2020] to retrieve code coverage and repeat creating test cases until we achieve for each subject system: (i) 70% of testing coverage and (ii) 40% of the mutants killed.

4.4 Collecting Metrics

To provide for a better overview of the subject configurable systems in our dataset, we collected a set of 14 software metrics. Table 4.1 shows these metrics. These metrics include traditional, CK, test, variability, and feature measurements, as shown in Figure 4.1. With METRICS [Metrics, 2020] and CK TOOL [CK, 2020], we compute traditional and CK metrics (e.g., number of lines of code and number of packages). With FEATUREIDE, we extract metrics related to variability (e.g., the number of features and valid configurations). As mentioned, with JACOCo [JaCoCo, 2020] and PIT [Coles et al., 2016], we retrieve metrics related to the test suite. To collect metrics related to each feature, we wrote our own script¹. For each system, our script searches for code snippets containing specific variability statements. Moreover, for each feature, it computes the number of classes, methods, constructors, and lines of code containing variability statements. We increase internal validity of our research through manually checking our script outcomes.

4.5 Test-enriched Configurable System Dataset

The 30 configurable systems of our dataset belong to several domains such as games, text editor, media management, and file compression. Tables 4.2 and 4.3 present an overview of these systems divided into size, variability, and test suite measures. Additional information of the dataset is available in our supplementary website [Ferreira et al., 2020c].

Size measures. We selected systems from a large variation of size regarding their number of lines of code (*LOC*), packages (*NOP*), classes (*NOC*), and methods (*NOSM*). For instance, subject configurable systems vary from 189 lines of code (*BANKACCOUNT*) to more than 150 000 lines of code (*ARGOUML*). Similarly, while *INTEGERSETSPL* has only 3 classes, *ARGOUML* has almost 2000 classes.

Variability measures. We selected systems with different variability. For instance, while *CHECKSTYLE* has 141 features, *CHESSE*, *TASKOBSERVER*, and *TELECOM* have only three features (*NOFE*) each. We can also see a similar variation in the number of valid configurations (*NCF*). For instance, while *ELEVATOR* has 20 valid configurations, *FEATUREAMP3* has 20 500 valid configurations.

Test suite measures. In the *FTC* column of Table 4.3, we see the number of test cases for each system of our dataset and the percentage increase compared to the initial

¹<https://github.com/fischerJF/Community-wide-Dataset-of-Configurable-Systems/tree/master/Parser>

Table 4.1: Dataset metrics, adapted from Metrics [2020]

Metric	Description
CBO (Coupling between Objects)	Counts the number of dependencies a class has
WMC (Weight Method Class)	Counts the number of branch instructions in a class
DIT (Depth Inheritance Tree)	Counts the number of fathers a class has
NOC (Number of Children)	Counts the number of children a class has
RFC (Response for a Class)	Counts the number of unique method invocations in a class
LCOM (Lack of Cohesion of Methods)	Measures how frequent methods in a class access common attributes
NOM (Number of Methods)	Counts the number of methods
NOPM (Number of Public Methods)	Counts only the number of public methods
NOC (Number of Class)	Counts the number of class
NOP (Number of Packages)	Counts the number of packages
NOSM (Number of Static Methods)	Counts the number of static methods in the selected scope
NOF (Number of Fields)	Counts the number of fields
NOFP (Number of Public Fields)	Counts only the public fields
NOSF (Number of Static Fields)	Counts only the static fields
NOSI (Number of Static Invocations)	Counts the number of invocations to static methods
LOC (Lines of Code)	Counts the lines of count, ignoring empty lines and comments
NOFE (Number of Features)	Counts the number of features
NCV (Number of Valid Configurations)	Counts the number of valid configurations
NVC (Number of Occurrences of Variability in the Source Code)	Counts the number of occurrences of variability in the source code
ITC (Initial Number of Test Cases)	Counts the initial number of test cases
FTC (Final Number of Test Cases)	Counts the final number of test cases
LTC (Lines of Testing Code)	Counts the number Lines of testing code
CV (Percentage of Test Suite Coverage)	Counts the percentage of test suite coverage
KM (Percentage of killed Mutants)	Counts the percentage of killed mutants

number of test cases (*ITC* column). Note that the number of lines of testing code varies from 207 for FEATUREAMP6 to 17 014 for ARGOUML. As expected, smaller systems often have fewer lines of testing code and test cases. At the end, our dataset has a total of 3 182 test cases of which we created 727 test cases for 20 configurable systems, 90 tests for the three systems we extended test suites and, the remaining 2 365 test cases come from systems that already had test suite.

4.6 Threats to Validity

Even with the careful planning, this research can be affected by different factors which might threaten our findings. We discuss below these factors and the main decisions we have made to mitigate their impact on the research.

We followed a careful set of procedures to create a repository of configurable systems and their test cases. As the number of open source configurable systems found is limited, and due to limited resources and the high effort required, we could not create a large repository with a high number of configurable systems. This limitation

Table 4.2: Size and variability metrics of the dataset

System Name	Size Measures				Variability Measures			
	LOC	NOP	NOC	NOM	NOFE	NCV	NVC	T
ATM [Santos et al., 2016]	1160	2	27	100	7	80	44	CC
ArgoUML [Martinez et al., 2017]	153977	92	1812	13034	8	256	1388	CC
BankAccount [SPL2go, 2020]	189	3	9	22	10	144	13	A
Checkstyle [Wong et al., 2018]	61435	14	78	719	141	$>2^{135}$	180	VE
Chess [Santos et al., 2016]	2149	7	22	162	3	8	20	CC
Companies [Souto et al., 2017]	2477	16	50	244	10	192	255	VE
Elevator [Meinicke et al., 2014]	426	2	7	59	5	20	9	VE
Email [Souto et al., 2017]	429	3	7	49	8	40	30	VE
FeatureAMP1 [SPL2go, 2020]	1350	4	15	93	28	6732	40	A
FeatureAMP2 [SPL2go, 2020]	2033	3	14	167	34	7020	55	A
FeatureAMP3 [SPL2go, 2020]	2575	8	16	223	27	20500	93	A
FeatureAMP4 [SPL2go, 2020]	2147	2	57	203	27	6732	57	A
FeatureAMP5 [SPL2go, 2020]	1344	3	9	895	29	3810	36	A
FeatureAMP6 [SPL2go, 2020]	2418	8	30	202	38	21522	76	A
FeatureAMP7 [SPL2go, 2020]	5644	3	46	220	29	15795	57	A
FeatureAMP8 [SPL2go, 2020]	2376	2	6	106	27	15708	48	A
FeatureAMP9 [SPL2go, 2020]	1859	3	8	134	24	6732	53	A
GPL [Souto et al., 2017]	1235	3	17	78	13	73	59	VE
IntegerSetSPL [SPL2go, 2020]	200	2	3	20	3	2	7	A
Jtopas [Souto et al., 2017]	4397	7	43	472	5	32	10	VE
MinePump [SPL2go, 2020]	244	2	7	26	7	64	4	A
Notepad [Souto et al., 2017]	1564	4	17	90	17	256	24	VE
Paycard [SPL2go, 2020]	374	2	8	27	4	6	10	CC
Prop4J [SPL2go, 2020]	1138	2	15	90	17	5029	17	A
Sudoku [Souto et al., 2017]	949	2	13	51	6	20	53	VE
TaskObserver [Santos et al., 2016]	486	2	10	33	4	8	9	CC
Telecom [Santos et al., 2016]	273	2	40	11	3	4	6	CC
UnionFindSPL [SPL2go, 2020]	335	2	36	5	13	10	12	A
VendingMachine [Martinez et al., 2017]	472	2	7	21	8	256	7	CC
ZipMe [Souto et al., 2017]	4647	3	311	33	13	24	343	VE

LOC: number of lines of code, *NOP*: number of packages, *NOC*: number of classes, *NOM*: number of methods, *NOFE*: number of features, *NCV*: number of valid configurations, *NVC*: Number of occurrences of variability in the source code, *T*: variability technique being possible variability encoding (VE), AHEAD (A), and conditional compilation (CC).

has implications in the amount of analyzed test faults, which is particularly relevant to the analysis of in Chapter 7. To mitigate this threat, we used a dataset proposed in a previous study that has configurable systems of different sizes, and we aim to cover at least 70% of the systems code and to kill at least 40% of the generated mutants.

Some factors may threaten the generalization of our results. For instance, it is not possible to ensure that the selected systems reflect the best samples of the recurrent practices. We mitigate this threat by searching for representative samples in the relevant literature. We also selected configurable systems from well-known repositories, such as SPL2Go [SPL2go, 2020]. In addition, we have also filtered systems with less than 1 000 lines of code since we considered them as toy examples. As another external validity threat, all systems that compose our dataset are developed in Java. Therefore,

Table 4.3: Test suite metrics of the dataset

System Name	ITC	FTC	LTC	CV	KM	TS
ATM [Santos et al., 2016]	0	76 (100%)	1371	91%	79%	C
ArgoUML [Martinez et al., 2017]	1326	1326 (0%)	17014	17%	9%	O
BankAccount [SPL2go, 2020]	0	42 (100%)	539	92%	62%	C
Checkstyle [Wong et al., 2018]	719	719 (0%)	13606	38%	5%	O
Chess [Santos et al., 2016]	0	77 (100%)	1296	72%	72%	C
Companies [Souto et al., 2017]	42	42 (0%)	1850	70%	46%	O
Elevator [Meinicke et al., 2014]	0	59 (100%)	683	92%	73%	C
Email [Souto et al., 2017]	30	85 (65%)	1429	97%	61%	E
FeatureAMP1 [SPL2go, 2020]	0	18 (100%)	977	85%	46%	C
FeatureAMP2 [SPL2go, 2020]	0	18 (100%)	698	72%	43%	C
FeatureAMP3 [SPL2go, 2020]	0	15 (100%)	725	77%	42%	C
FeatureAMP4 [SPL2go, 2020]	0	12 (100%)	622	82%	40%	C
FeatureAMP5 [SPL2go, 2020]	0	17 (100%)	730	91%	49%	C
FeatureAMP6 [SPL2go, 2020]	0	9 (100%)	207	31%	43%	C
FeatureAMP7 [SPL2go, 2020]	0	8 (100%)	180	28%	40%	C
FeatureAMP8 [SPL2go, 2020]	0	78 (100%)	1637	82%	42%	C
FeatureAMP9 [SPL2go, 2020]	0	105 (100%)	1975	83%	63%	C
GPL [Souto et al., 2017]	45	51 (12%)	1162	83%	60%	E
IntegerSetSPL [SPL2go, 2020]	0	19 (100%)	286	100%	80%	C
Jtopas [Souto et al., 2017]	87	87 (0%)	6703	67%	50%	O
MinePump [SPL2go, 2020]	0	34 (100%)	459	91%	65%	C
Notepad [Souto et al., 2017]	25	25 (0%)	1790	59%	15%	O
Paycard [SPL2go, 2020]	0	13 (100%)	453	88%	61%	C
Prop4J [SPL2go, 2020]	63	63 (0%)	504	71%	67%	O
Sudoku [Souto et al., 2017]	6	35 (82%)	650	80%	67%	E
TaskObserver [Santos et al., 2016]	0	24 (100%)	280	91%	71%	C
Telecom [Santos et al., 2016]	0	26 (100%)	391	99%	65%	C
UnionFindSPL [SPL2go, 2020]	0	40 (100%)	616	84%	66%	C
VendingMachine [Martinez et al., 2017]	0	37 (100%)	297	97%	83%	C
ZipMe [Souto et al., 2017]	22	22 (0%)	703	41%	19%	O

ITC: initial number of test cases, *FTC*: Final number of test cases, *LTC*: Lines of testing code, *CV*: percentage of test suite coverage, *KM*: percentage of killed mutants, *TS*: test suite being created by us (C), extended by us (E), or original (O).

we cannot claim that similar results would have been observed in other programming languages or technologies. This is a common limitation of several research studies on configurable software systems [Kim et al., 2012b, 2013; Meinicke et al., 2014; Souto et al., 2017; Wong et al., 2018].

Our findings are also restricted to the set of selected metrics, such as source code and test metrics. We quantify eight metrics for all configurable systems that compose our repository and we also compute metrics for test and feature characteristics. To make these measurement processes easier and automated, we used specific tools, such as CK Tool [CK, 2020] and JaCoCo [JaCoCo, 2020]. In addition, we also manually checked few cases to ensure the results of these tools were as expected, since some used tools were not aimed at measuring code of configurable systems. Therefore, we believe that similar conclusions would also be achieved if someone uses different measures and tools that quantify similar attributes or features for this set of configurable systems.

4.7 Final Remarks

To create a test-enriched dataset, we searched for configurable software systems in the literature. As a result, we found 243 systems being 60 developed in Java-based programming languages. However, we only found 10 systems with a test suite available. To increase this number, we created a test suite for other 20 projects. As means for quality assurance, we created tests until they have a code coverage of 70% and kill at least 40% of mutants. The final dataset has 30 systems varying in domains, size, variability, and test suite size. We provide three groups of metrics (traditional, variability, and test suite) to characterize the proposed dataset. We are confident this dataset will benefit practitioners and also be useful for researchers comparing testing approaches and methods.

Several datasets for the configurable systems have been proposed and used before. However, this dataset is the first dataset for configurable systems with an extensive test suite [Ferreira et al., 2020b]. We believe our dataset can be a common point of comparison for configurable system testing strategies. Researchers, testers, and developers can benefit from our dataset of configurable systems with an extensive test suite. In the next two chapters, we present two empirical studies with sound and t-wise testing strategies.

Chapter 5

Sound Testing Tools: A Comparative Study

Testing configurable software systems is challenging due to the number of configurations to run with each test, leading to a combinatorial explosion in the number of configurations and tests. Currently, several testing techniques and tools have been proposed to deal with this challenge, but their potential practical application remains mostly unexplored. The lack of studies to explore the tools motivated us to design and perform a comparative empirical study of the two sound testing tools named *VarexJ* [Meinicke et al., 2016] and *SPLat* [Kim et al., 2013]. This sound testing tool was found in the systematic mapping study (see Chapter 3). They are considered sound testing techniques [Souto et al., 2017; Liebig et al., 2013b; Meinicke et al., 2016; Kim et al., 2012a, 2013] because they explore all reachable configurations from a given test.

Our main goal is to identify the advantages and drawbacks of these testing tools. Therefore, we analyzed the effectiveness and efficiencies of *VarexJ* and *SPLat* to test a group of configurable systems from our dataset presented in the Chapter 4. In summary, *VarexJ* [Meinicke et al., 2016] is a variability-aware interpreter for Java bytecode and *SPLat* [Kim et al., 2013] explores all reachable configurations from a given test. Therefore, researchers and testers can benefit from our empirical study. We believe that researchers can observe the benefits and limitations of the analyzed tools. Testers can, through our comparative study, choose the most appropriate tool to test their configurable systems. For more details see [Ferreira et al., 2019].

The remainder of this chapter is organized as follows. Section 5.1 presents the research questions. Section 5.2 describes the steps of the study. Sections 5.3 and 5.4 discusses our achieved result. Section 5.5 discusses the threats to the validity of the study. Finally, Section 5.6 concludes this chapter.

5.1 Goal and Research Questions

We conducted a comparative study to evaluate two testing tools found in the systematic mapping studies described in Chapter 3. We systematically defined our goal based on the goal question metric (GQM) template [Basili and Rombach, 1988]. We *analyze* two sound testing techniques *for the purpose of* identify advantages and drawbacks of testing tools that use sound testing techniques; *with respect to* support practitioners and researchers to choose the most appropriate strategy to fulfill their needs, avoid the recurrence of similar faults, and improve existing testing strategies; *from the viewpoint of* researchers and software developers with expertise in software testing *in the context of* same target systems and test suites in the execution of each tool. The idea is to compare the faults returned by both tools (*VarexJ* and *SPLat*). We perceived a lack of studies that evaluate testing tools for configurable systems concerning the effectiveness of finding faults. To support our empirical study, we defined the following research questions.

RQ₁ *How efficient are the sound tools for testing configurable systems?* RQ₁ investigates how much time is spent by the tools to test our set of selected projects. This is an important direction since the cost to run the test suite for configurable systems is a critical point.

RQ₂ *How effective are the sound tools for testing configurable systems?* We propose RQ₂ to investigate the effectiveness of sound tools for detecting configurable systems faults.

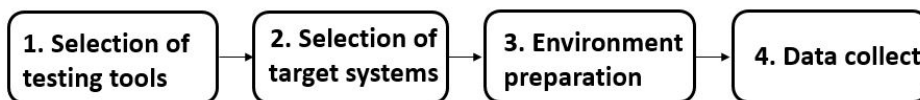


Figure 5.1: Steps of the empirical study

5.2 Study Steps

To answer the research questions presented in Section 5.1, we evaluate two testing tools for configurable systems named *VarexJ* and *SPLat*. Figure 5.1 shows the four study steps we followed in the proposed empirical study. Step 1 is the filtering of selected testing tools. We show the selection criteria adopted in this step. Step 2 presents the target systems used to compare the testing tools. In addition, we demonstrate the

empirical study environment in Step 3. Finally, Step 4 consists of running the selected tools and collecting data of interest. These four steps are detailed below.

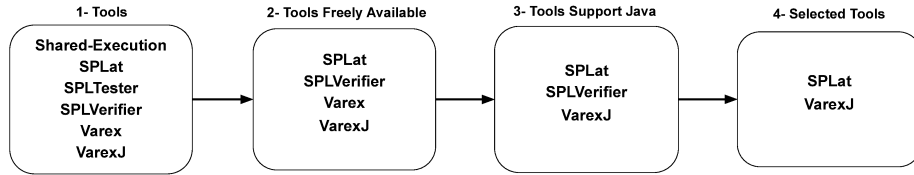


Figure 5.2: Select of testing tools

Selection of testing tools. We found 34 testing tools for configurable systems in the systematic mapping study reported in Chapter 3. After applying a set of filtering criteria, we conducted a study with *VarexJ* [Meinicke et al., 2016] and *SPLat* [Kim et al., 2013]. The selection process was conducted as follows. Figure 5.2 shows the steps of the filtering process for the tools found. First, we focus our study in the Java programming language to study, since the majority of the testing tools analyses configurable systems developed in this language. We adopted the criteria to use tools which claim to test all possible configurations at the source code level of the configurable systems (Sound testing techniques). We obtained six tools: Shared-Execution [Kim et al., 2012a], *SPLat* [Kim et al., 2013], *SPLTester* [Kim et al., 2011b], *SPLVerifier* [Apel et al., 2011], *Varex* [Nguyen, 2014], and *VarexJ* [Meinicke et al., 2016] (Figure 5.2 Step 1). Then, we restricted the set of tools to include only tools freely available (Figure 5.2 Step 2). After applying these criteria, we end up with 3 out of 6 tools previously reported (Figure 5.2 Step 3). In our study, we discarded *SPLVerifier* [Apel et al., 2011] because it was not able to automatically switch configurations in the selected configurable system. To use *SPLVerifier*, one must first inform the configurations to be tested as individual products. Therefore, we selected two testing tools for configurable systems in this comparative study: *SPLat* [Kim et al., 2013] and *VarexJ* [Meinicke et al., 2014] (Figure 5.2 Step 4). A brief description of *SPLat* and *VarexJ* tools is presented below.

SPLat implements a dynamic technique that explores all reachable configurations from a given test, by monitoring feature variable accesses during test execution and, based on that, deciding which configurations should be executed. As output, *SPLat* returns, for each test, the configurations explored and the respective results (pass or fail).

VarexJ is a variability-aware interpreter for Java bytecode [Meinicke et al., 2016]. It executes all reachable configurations of a configurable system in a single run, avoiding redundant executions by means of sharing calculations and aggregated results. As

output, *VarexJ* logs the JVM stack trace of each failure and gives boolean expressions representing combination (or interaction) of features that lead to each one.

Selection of the target systems. We selected seven configurable systems of our dataset (Chapter 4) to compose our empirical study. Due to the limitation of the testing tools analyzed, we discarded systems that had a graphical interface and had more than 17 features. In this way, the configurable systems used in this chapter were COMPANIES, ELEVATOR, EMAIL, GPL, JTOPAS, SUDOKU, and ZIPME. These target systems were already analyzed in previous studies [Kim et al., 2013; Meinicke et al., 2016; Souto et al., 2017].

Environment preparation. We installed the selected tools with all dependencies in a personal computer prepared for the study. This computer has 16 GB of RAM, processor i7 3.60 GHz and operating system Windows 10. To use the *VarexJ* and *SPLat* tools in our research, we instrumented the source code and test suite of the target systems at each point of variability. The two tools used in the study rely on the feature model to validate possible combinations of features found in the tests. We had to translate the feature model into the format suitable to *VarexJ*, the *DIMACS* Conjunctive Normal Form format. However, we do not need to translate the feature model into the *SPLat* format, which is the *Guidsl Grammar* [Guidsl, 2020] because all target systems already used these files.

We also need to know the maximum time limit to run a test suite for every valid configuration on all target systems. To achieve this goal, we developed a baseline algorithm that finds all valid configurations and runs the test suite for each of them. For this, the baseline implemented an algorithm for generating power sets. We used this implementation by providing a set of features (of each target configurable systems) as input and retrieving all possible combinations of them as output.

Although the power set algorithm has exponential complexity and does not scale to a large number of features, it was possible to use this approach to find all possible feature combinations of each target configurable systems because the systems are small. We used a feature model validator to check each possible configuration returned by the power set. We then ran the test suite for each target configurable systems only for the valid configurations¹.

Data collection. This step consists of running the selected testing tools against the test suite of the target systems and collecting the data to evaluate such tools in terms of efficiency (RQ₁) and effectiveness (RQ₂). For the former evaluation, we mean the execution time. To mitigate random effects, we conducted the experiments running

¹https://github.com/fischerJF/Community-wide-Dataset-of-Configurable-Systems/tree/master/workspace_IncLing/Baseline

each tool against each target system test suite 100 times and computed the average execution time. For the effectiveness evaluation, we measure the amount of configurations causing faults that each testing tool was able to identify. For each target system, we followed the process depicted in Figure 5.3 to retrieve data for measuring the effectiveness of *VarexJ* and *SPLat*. The log generated by both testing tools provide similar pieces of information, corresponding to a general set of configurations that causes a fault (e.g., all configurations with features X and Y enabled) and the respective stack traces. We first parsed the text log into a structured text containing the representation of configurations that cause the faults and the respective lines of test cases from which the faults emerged.

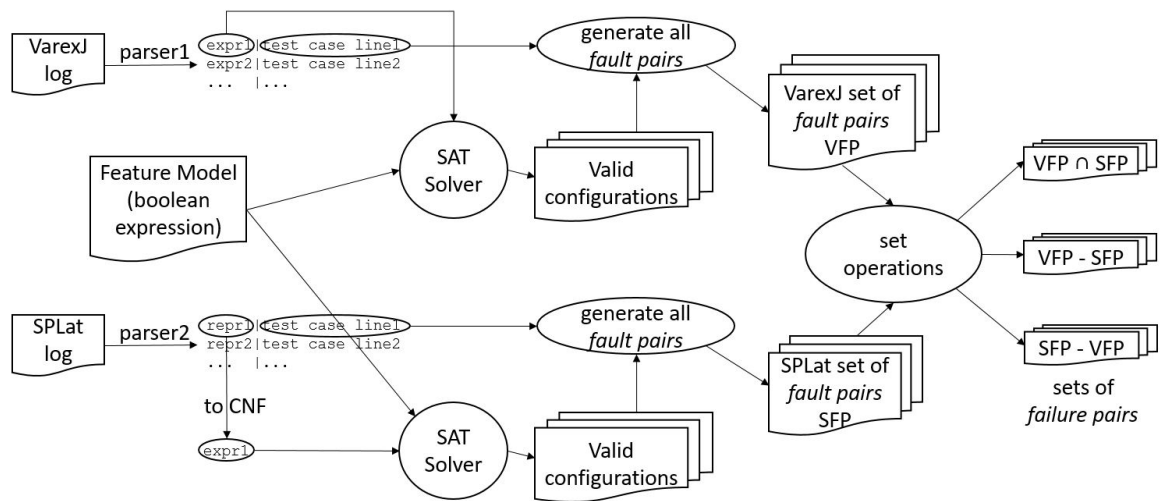


Figure 5.3: Effectiveness data collection process

Apart from this process, we previously translated the feature model of each target system into a boolean expression. With the aid of a SAT Solver, takes as inputs such feature model and a configuration set also represented as a boolean expression. We retrieved all valid configurations that caused faults and mapped them to each line of the test cases. We named such information data as *fault pair*, which is composed by a valid configuration and a test case. The more fault pairs a tool is able to find, the more effective it is. Finally, we performed set operations with the sets of fault pairs obtained for *VarexJ* (*VFP*) and for *SPLat* (*SFP*), generating three other sets: $VFP \cap SFP$ (pairs found by *VarexJ* and *SPLat*), $VFP - SFP$ (pairs found by *VarexJ* and not by *SPLat*), and $SFP - VFP$ (pairs found by *SPLat* and not by *VarexJ*).

Table 5.1: Execution time

Target Systems	#features	#Confs.	Baseline (ms)	<i>SPLat</i> (ms)	<i>VarexJ</i> (ms)
JTopas	6	16	57	1443	412
Elevator	6	20	170	377	152
Sudoku	7	20	87	269	2479
Email	9	40	369	4703	33023
Companies	10	96	832	2518	360
GPL	13	124	6999	3647	1196
ZipMe	13	48	11347	208	466

#features: Feature set that the target system has; **#Confs.:** Total valid configurations;

Baseline: Time in ms returned by baseline algorithm; ***SPLat*:** Time in ms returned by *SPLat* tool

***VarexJ*:** Time in ms returned by *VarexJ* tool.

5.3 Efficiency of the Tools

RQ₁ *How efficient are the sound tools for testing configurable systems?*

Table 5.1 shows the collected data of the time spent by the *VarexJ* and *SPLat* tools to run the target systems. We observe *SPLat* spent significantly less time to run the target systems EMAIL and SUDOKU. In turn, considering JTOPAS, COMPANIES, and GPL, *SPLat* spent more time than *VarexJ*. For EMAIL, we noticed that one of its test classes has a particular `setUp` method that is executed before each of its test cases that caused the execution time in *VarexJ* much higher than in *SPLat*. This method resets an array of `Client` objects, setting `null` to all of its 2,000,000 positions. Since *VarexJ* implements context-aware and aggregated results, the structure of such array causes a considerable overhead for the reset operation.

For the SUDOKU target system, *VarexJ* spent more time than *SPLat* to test it. The main reason for this is because SUDOKU manipulates files for the construction of its game board and *VarexJ* also takes a considerable overhead for file manipulation. In addition, the overhead to start the execution is high in *VarexJ* and, since SUDOKU has only 20 valid configurations to be explored, *SPLat* executes faster this configurable system test suite. For the COMPANIES and GPL target systems, *SPLAT* spent more testing time compared to *VarexJ* because of the number of features and products that these systems have. As can be seen in Table 5.1, COMPANIES and GPL have 10 and 13 features, and 96 and 124 different valid configurations, respectively. Since the number of variation points in the code is high to test all possible configurations achievable by the test cases, *VarexJ* can execute faster than *SPLat*.

Table 5.2: Measurement of effectiveness

Target Systems	<i>VarexJ</i>	<i>SPLat</i>	<i>VarexJ</i> - <i>SPLat</i>	<i>SPLat</i> - <i>VarexJ</i>	<i>VarexJ</i> \cap <i>SPLat</i>
JTopas	16	28	16	28	0 (0%)
Elevator	0	0	0	0	0 (0%)
Sudoku	82	48	34	0	48 (58%)
Email	44	43	1	0	43 (98%)
Companies	701	600	202	101	499 (62%)
GPL	731	589	335	193	396 (42%)
ZipMe	600	588	276	264	324 (37%)

VarexJ: Number of pairs found by *VarexJ*; **#*SPLat***: Number of pairs found by *SPLat*;

VarexJ* - *SPLat: Number of pairs found by *VarexJ* and not found by *SPLat*;

SPLat* - *VarexJ: Number of pairs found by *SPLat* and not found by *VarexJ*;

VarexJ* \cap *SPLat: Number of pairs found by both tools (intersection).

Additionally, Table 5.1 shows baseline execution times which grows according to the number of features, since constructing the power set of the features is the greater computational cost for the baseline. For the *VarexJ* and *SPLat* tools, we have an initial overhead. However, according to Table 5.1, *SPLat* and *VarexJ* can have a smoother time growth than the baseline. *SPLat* and *VarexJ* are able to cope better with the growth in the number of configurations, due to their strategies in testing valid configurations.

RQ₁ Summary. We note that *VarexJ* is generally more efficient than *SPLat*. However, when it was not more efficient, it was by a large difference in specific situations related to its implementation of variability-aware execution.

5.4 Effectiveness of the Tools

RQ₂ *How effective are the sound tools for testing configurable systems?*

The first and second columns of Table 5.2 indicate the total fault pairs found by *VarexJ* and by *SPLat*, respectively. The fourth and fifth columns of Table 5.2 indicate total fault pairs found by *VarexJ* and not by *SPLat*, and total fault pairs found by *SPLat* and not by *VarexJ*, respectively. Finally, the last column indicates the total number of fault pairs that were found by both tools.

Apart from ELEVATOR, *VarexJ* and *SPLat* found feature interaction faults in all of the target systems. Consequently, we got fault pairs for six systems. Given the characteristics of both *VarexJ* and *SPLat* regarding their implementations of STT, we were expecting more similar effectiveness results for those tools, i.e., higher intersection

values. However, Table 5.2 usually shows the opposite. The higher intersection values found were for the systems: EMAIL, COMPANIES, and SUDOKU. As can be seen the differences presented in Columns 4 and 5 of the Table 5.2. Despite that, for those 3 target systems, we verified that *VarexJ* found more fault pairs than *SPLat*.

VarexJ and *SPLat* found faults in distinct test cases for JTOPAS. Consequently, they got distinct fault pairs, which led to an empty intersection set of fault pairs for this target system. Moreover, JTOPAS was the only system for which *SPLat* found more fault pairs than *VarexJ*. For SUDOKU, *VarexJ* found all fault pairs found by *SPLat* and more 34. Finally, for EMAIL, *VarexJ* found only one fault pair that *SPLat* did not. Indeed, both tools found faults with the same two test cases. Interestingly, however, from *VarexJ* log, we found an error in a configuration in which all optional features were disable, whereas *SPLat* log did not provided that.

RQ₂ Summary. We observed that *VarexJ* and *SPLat* presented distinct results for efficiency while testing the target systems. Although *VarexJ* found more faults than *SPLat* for the majority of the target systems, such result deserves a more in-depth investigation because we expected a higher intersection of faults encountered by them.

5.5 Threats to Validity

A key issue in empirical studies like study presented in this chapter is the validity of the results. Even with careful planning, different factors may affect these research results [Wohlin, 2014]. This section presents potential threats to the study validity and discuss some bias that may have affected the study results. We also explain our actions to mitigate them.

Conclusion Validity. We extracted fault pairs as the unit of measurement for effectiveness evaluation (see Section 5.2). This data extraction could, if misconducted, lead us to incorrect results and conclusions. To mitigate this threat, two researchers manually analyzed the fault pairs and attempt to infer similar information from the log provided by each evaluated testing tool. It was done with all the pairs generated for JTOPAS, SUDOKU, and EMAIL, and with a sampling of the pairs generated for COMPANIES, GPL and ZIPME.

External Validity. We cannot claim that our results directly generalize to other environments and tools, such as to industry practices. A major threat to validity in this case can be the selected tools and the target systems. We choose two tools classified

as sound testing techniques and we cannot guarantee that our observations can be generalized for this category. We tried to minimize this threat choosing tools that implement different testing techniques, such as variability-aware execution (*VarexJ*) and a lightweight dynamic analysis for reducing combinatorics (*SPLat*).

Construction Validity. We translate the feature model of the target systems for Conjunctive Normal Form (CNF) that is the way *VarexJ* reads a feature model. This may be a threat to the construction validity because the translation may contain mistakes. To mitigate this threat, we made a truth table for each type and compared the results. We considered the correct translations when the truth tables returned the same results.

External Validity. First, we have performed our study with seven configurable systems. As one could expect, our target systems may not represent the characteristics of all configurable systems. To mitigate the effect of the configurable systems' representativeness chosen to compose our study, we are confident that we selected systems from various domains and test suite sizes. Second, we have restricted our analysis to configurable systems developed in the JAVA programming languages. Thus, we cannot generalize to other programming languages. This limitation may affect the generalization of our results. Finally, the threat to our study is the quality of the test suite in the selected configurable systems. All of our analysis is conditioned on the test suite's ability to reveal faults. To mitigate the test to be different for the SPLAT to VarexJ, we have removed from our study all tests of the target systems that exercise graphical interface and manipulation of files because VarexJ presented some complex issues such as execution abortion.

5.6 Final Remarks

This chapter presented a comparative empirical study with two tools (*SPLat* and *VarexJ*) selected from the systematic mapping study (Section 3). The goal was to compare these two tools when testing all valid configurations of the configurable systems. This chapter's main contribution was to provide researchers and testers with an analysis of the two strategies' effectiveness and efficiency. Regarding the effectiveness of *SPLat* and *VarexJ*, we note that *VarexJ* is generally more efficient than *SPLat*. However, when it was not more efficient, it was by a large difference in specific situations related to its implementation of variability-aware execution. Furthermore, for efficiency, we observed that *VarexJ* and *SPLat* presented distinct results while testing the target systems. Although *VarexJ* found more faults than *SPLat* for the majority

of the target systems, such result deserves a more in-depth investigation because we expected a higher intersection of faults encountered by them.

In the next chapter, we present a comparative empirical study with eight sampling testing strategies. This comparison aims to find which strategies are faster, more comprehensive, effective on identifying faults. We also analyse the reasons why a strategy performed better in one investigated property.

Chapter 6

Evaluating T-wise Testing

In the previous chapter, we presented a study with two main sound testing tools namely *VarexJ* [Meinicke et al., 2016] and *SPLat* [Kim et al., 2013]. However, the extensively testing of all valid configurations may be infeasible in practice and several sampling testing strategies have been proposed to recommend an optimal sample of configurations able to find most existing faults. In this chapter, we use our dataset proposed in Chapter 4 and compare recommended configurations from sixteen t-wise testing strategies (e.g., *ICPL-T2* and *IncLing -T2*). This comparison aims to find which strategies are faster, more comprehensive, effective on identifying faults, time-efficient, and coverage-efficient in our dataset and the reasons why a strategy fared better in one investigated property.

T-wise interaction sampling defines a set of a cost-effective sampling techniques for discovering interaction faults in configurable systems [Henard et al., 2014a]. These techniques have attracted the interest of several researchers because they achieve effective results with lower cost by minimizing the number of configurations to be tested even when using small values for t (e.g., 1 or 2) [Al-Hajjaji et al., 2016a; Garvin et al., 2011; Henard et al., 2014a; Johansen et al., 2011; Kaltenecker et al., 2019; Krieter et al., 2020; Kuhn and Reilly, 2002; Nie and Leung, 2011; Xiang et al., 2021]. In this work, we chose to evaluate t-wise strategies implemented in FeatureIDE [Thüm et al., 2014].

Our goal in this chapter is to provide a comparison of sampling testing strategies. The comparison of testing strategies may benefit practitioners supporting their choice of a testing strategy that best fits their needs. On the other hand, this study may also benefit researchers and tool builders by showing them opportunities for improving existing testing strategies and tools. To achieve our first goal, we choose (i) a dataset, (ii) testing strategies to be compared, and (iii) the comparison criteria. The dataset used in this evaluation is composed of all configurable software systems presented in

Chapter 4. As testing strategies to be compared, we selected variations of five t-wise sampling testing strategies: *CASA* [Garvin et al., 2011], *Chvatal* [Johansen et al., 2012b], *ICPL* [Johansen et al., 2011], *IncLing* [Al-Hajjaji et al., 2016a], and *YASA* [Krieter et al., 2020]. We selected these strategies because t-wise strategies assure a degree of testing coverage on the recommended configurations (see details of our choice in Section 6.2). At the end, we compare sixteen t-wise strategies (*CASA-T1*, *CASA-T2*, *CASA-T3*, *CASA-T4*, *Chvatal-T1*, *Chvatal-T2*, *Chvatal-T3*, *Chvatal-T4*, *ICPL-T1*, *ICPL-T2*, *ICPL-T3*, *IncLing-T2*, *YASA-T1*, *YASA-T2*, *YASA-T3*, and *YASA-T4*) and two baselines (brute force and random selection). As comparison criteria, we evaluate which testing strategies are faster, more comprehensive (i.e., with greater coverage), more effective in identifying faults, time-efficient, and coverage-efficient. To make our study feasible, we limit the number of recommended configurations for the baselines for up to 250 configurations.

The remainder of this chapter is organized as follows. Section 6.1 presents our goal and research questions. Section 6.2 describes the subject testing strategies. Sections 6.3 and 6.4 describe how we acquire data and operationalize the answer of our research questions, respectively. Sections 6.5 to 6.9 present the results achieved. Sections 6.10 to 6.12 discusses the results. Section 6.13 describes the main limitations and threats to validity of this study. Finally, Section 6.14 concludes this chapter.

6.1 Goal and Research Questions

Based on the goal question metric (GQM) template [Basili and Rombach, 1988], we systematically defined our goal. We *analyze* sixteen testing strategies *for the purpose of* identifying the fastest, most comprehensive, most effective, and most efficient strategies; *with respect to* support practitioners and researchers to choose the most appropriate strategy to fulfill their needs, avoid the recurrence of similar faults, and improve existing testing strategies; *from the viewpoint of* researchers and software developers with expertise in software testing *in the context of* a previously proposed dataset (Chapter 4) of configurable software systems with test suite available. Previous work [Engström and Runeson, 2011; Lopez-Herrejon et al., 2015; Machado et al., 2014] reported a lack of empirical evaluation based on a community-wide dataset to guide the comparison of different testing strategies. Motivated by this goal, we formulate the following research questions.

RQ₁: *What are the fastest strategies for testing configurable systems?*

RQ1.1: Which are the fastest testing strategies on generating a list of configurations?

RQ1.2: Which configurations suggested by testing strategies do execute faster?

RQ2: Which testing strategies do suggest a list of configurations that covers most configurations in configurable systems?

RQ3: Which testing strategies are more effective on finding faults in configurable systems?

RQ4: Which testing strategies are more time-efficient on finding faults in configurable systems?

RQ5: Which testing strategies are more coverage-efficient on finding faults in configurable systems?

These research questions show which testing strategies are faster, more comprehensive, more effective, and more efficient. This comparison with a community-wide dataset may benefit software testers because, from now on, they have results for several configurable systems to support their choice of a testing strategy that best fits their needs. Note that in the first two research questions, we provide a broader view of the configurable systems in the subject dataset. In the last three research questions, we look only at faulty systems.

6.2 Selected Testing Strategies

We focus on t-wise testing strategies because (i) they ensure certain quality of the set of suggested configurations (see Chapter 2), and (ii) the literature lacks a comparison among them on a community-wide dataset [Engström and Runeson, 2011; Lopez-Herrejon et al., 2015; Machado et al., 2014]. We are confident that we selected well-known testing strategies, once all selected strategies are developed in the FEATUREIDE - an integrated development environment (IDE) widely used by developers of configurable systems [Thüm et al., 2014]. Some strategies like *Chvatal* presents versions for 1-, 2-, 3-, and 4-wise. However, other strategies, such as *IncLing*, are only available for 2-wise tests. We investigate all testing strategies versions available in FEATUREIDE. The constraint for 4-wise is due to the number of suggested configurations. That is, the running time would increase significantly and make our study unfeasible.

At the end, we compared 16 t-wise strategies and two baselines. The baselines are mainly important to identify faults in the subject systems. For short, we selected

brute force (baseline 1), *random* (baseline 2), four variations of *CASA*, four variations of *Chvatal*, three variations of *ICPL*, one variation of *IncLing*, and four variations of *YASA*. Next, we briefly present all testing strategies investigated.

Brute Force (baseline 1) [Al-Hajjaji et al., 2016b; Thüm et al., 2014] is a strategy that generates all distinct valid configurations. However, due to time constraints, we generate a fixed number of valid configurations for a configurable system (i.e., 250).

Random (baseline 2) [Al-Hajjaji et al., 2016b; Thüm et al., 2014] is a SAT solver-based strategy that randomly generates a pre-defined number of valid configurations.

CASA [Garvin et al., 2011] is a greedy algorithm for sampling test configurations (more specifically, a simulated annealing algorithm). It works on two iterative steps. First, it minimizes the number of created configurations. Second, it ensures that a certain degree of coverage is achieved. We use *CASA* versions 1-, 2-, 3-, and 4-wise.

Chvatal [Johansen et al., 2011] is an adaptation of a greedy algorithm proposed by *Chvatal* [Chvatal, 1979] to solve the covering array problem. At the end, it is a heuristic that selects a subset of possible configurations with a t-wise covering array. We use *Chvatal* versions 1-, 2-, 3-, and 4-wise.

ICPL [Johansen et al., 2012b] is an algorithm for t-wise covering arrays. This strategy is also based on the *Chvatal* algorithm [Chvatal, 1979]. However, it contains optimizations for increasing its performance. We use *ICPL* versions 1-, 2-, and 3-wise.

IncLing [Al-Hajjaji et al., 2016a] is an incremental sampling for 2-wise (i.e., pair-wise) interaction testing. The main difference between *IncLing* and other testing strategies is that *IncLing* generates configurations one at a time to enhance sampling efficiency in terms of interaction coverage rate.

YASA [Krieter et al., 2020] is based on the traditional *IPOG* algorithm [Lei et al., 2008; Yu et al., 2013], which starts with a given empty sample and then iterates over all t-wise once at the time. Through the application of different heuristics and caching methods, this testing strategy, in theory, improves its sampling time compared to other t-wise sampling strategies. We use *YASA* versions 1-, 2-, 3-, and 4-wise.

6.3 Data Acquisition

Our data acquisition consists basically of three tasks: (i) run the testing strategies presented in Section 6.2 for each configurable system in the subject dataset (Chapter 4), (ii) extract information from logs creating a set of true faults (reference list), and (iii) collect metrics. In what follows, we give details on how we automated these tasks.

Running Testing Strategies. Once we run a testing strategy, it returns a list of configurations. Hence, for each configuration, we selected the set of features of the target configuration, ran the testing suite related to it, and analyzed the outcome logging. As it is a time-consuming and error-prone task, we created a script to automate this task. Considering time constraints, we defined an upper threshold of 250 configurations per subject system and testing strategy.

Creating a Reference List. We analyzed the execution log of the 18 (16 t-wise strategies and two baselines) testing strategies against the 30 subject systems. After parsing the log, we found test cases of which a fault emerged as well as the reason why it happened. Hence, we investigated each reported fault to confirm that it is truly a fault and if it arose due to a feature interaction. The reference list is the union of all faults found on all configurations suggested by all testing strategies investigated in this study.

Metrics Collection. We use different tools to extract metrics used in our study. Once we did not find tools to compute metrics used to answer our research questions, we computed them with our scripts (Chapter 4). Our analysis scripts (written in Java) are open-source. All data necessary for replicating this study are stored in CSV files. All tools, links to subject projects, reports of faults for each testing strategy, and data used in this study are available at our supplementary website [Ferreira et al., 2020c].

6.4 Operationalization

For each research question, we defined the following null and alternative hypotheses.

(H0) All testing strategies present similar results.

(H1) At least one testing strategy differ from the others.

To perform a normality test, we used the Shapiro-Wilk method [Razali and Wah, 2011]. As a result, this test failed in all cases indicating that our data do not follow a normal distribution. Taking this information into account, we used Friedman’s Test to verify the hypotheses formulated in our study since it is used for one-way repeated measure analysis of variance by ranks. For short, this test detects differences in treatments across multiple test attempts [Sheskin, 2020]. In our case, it is similar to the Kruskal–Wallis one-way analysis of variance by ranks. When it was possible to reject the null hypothesis, we used Post Hoc Analysis to identify which subject testing strategies tend to statistically differ from the others.

Next, we detail how we answered each research question. For the first five research questions, the independent variables are the 16 t-wise testing strategies.

Answering RQ₁. To answer RQ₁, we compute the time (in seconds) to generate configurations (RQ1.1) and the time (in seconds) to run the test suite for the configurations suggested by each subject testing strategy (RQ1.2). In summary, number of seconds (*#Seconds*) consists of the sum of the time to generate configurations and to execute the suggested configurations. To mitigate random effects on time measurement, we use the average of seconds, performing these analyses ten times. We use a computer with 16 GB of RAM, i7 processor 3.60 GHz, Windows 10, and JVM with 2 GB of memory. The lower the *#Seconds*, the faster the testing strategy is. We report results for each configurable system and show which strategies performed better for a greater number of configurable systems. In RQ₁, *#Seconds* is our dependent variable.

Answering RQ₂. To answer RQ₂, we compute the percentage of the number of configurations reported by a testing strategy (*#Configurations*) over the number of valid configurations for each configurable system. The higher the percentage, the higher the testing strategy coverage. We report results for each configurable system and also show which testing strategies are more comprehensive for a greater number of configurable systems. In RQ₂, *#Configurations* is our dependent variable.

Answering RQ₃. To answer RQ₃, we use *recall* (Eq. 6.1). In our context, recall is the number of correct faults found by the configurations suggested by a target testing strategy (i.e., true positive faults) divided by the number of existing faults in our reference list (i.e., the sum of true positive and false negative faults). The higher the recall, the more effective the testing strategy is. We report the recall calculated for each testing strategy over each configurable system from which at least one feature interaction fault exists. In RQ₃, *recall* is our dependent variable.

$$Recall = \frac{TP\ faults}{(TP\ faults + FN\ faults)} \quad (6.1)$$

Answering RQ₄. To answer RQ₄, we compute *time-efficiency* (Eq. 6.2). Time-efficiency is the number of correct faults found by the configurations suggested by a target strategy divided by *#Seconds* used to answer RQ₁. The higher the *time-efficiency*, the more efficient the strategies. We report results for each configurable system and show which strategies performed better for the higher number of systems. In RQ₄, *time-efficiency* is our dependent variable.

$$TimeEfficiency = \frac{TP\ faults}{\#Seconds} \quad (6.2)$$

Answering RQ₅. To answer RQ₅, we compute *coverage-efficiency* (Eq. 6.3). Coverage-efficiency is the number of faults found by the configurations suggested by a target strategy divided by *#Configurations* used to answer RQ₂. The higher the *coverage-efficiency*, the more efficient the testing strategy. We report results for each configurable system and show which testing strategies performed better for a higher number of configurable systems. In RQ₅, *coverage-efficiency* is our dependent variable.

$$CoverageEfficiency = \frac{TP\ faults}{\#Configurations} \quad (6.3)$$

6.5 The Fastest Testing Strategies (RQ1)

Table 6.1 presents the sum of the time to generate configurations (RQ1.1) and execute the test suite (RQ1.2) for all configurations suggested by each subject testing strategy. We highlight the time of the fastest testing strategy for each configurable system according to the t-wise group. Once multiple strategies have the same shortest time, we consider those as the fastest ones. Note that the time to generate configurations and execute these configurations varied from 3 to 28963 seconds (≈ 8 hours) and, in general, 1- and 4-wise strategies were respectively faster and slower than the other testing strategies. In addition, in most cases, the time to generate the configurations were far greater than the time to execute the configurations. As an extreme example, for FEATUREAMP9 using *CASA-T₄*, it was necessary 28800 seconds to generate the configurations and 163 seconds to execute the test suite for the suggested configurations.

As a result, we can see that for the 1-wise group, *ICPL-T1* is faster than the other testing strategies for ten configurable systems. *CASA-T1* and *YASA-T1* are the fastest for nine configurable systems. For the 2-wise group, *ICPL-T2* is faster than the other testing strategies for 12 configurable systems. *CASA-T2* is the fastest for ten configurable systems. *YASA-T2* is the fastest for seven configurable systems. For the 3-wise group, *ICPL-T3* is faster than the other testing strategies for 11 configurable systems. *CASA-T3* is the fastest for nine configurable systems. *YASA-T3* is the fastest for eight configurable systems. For the 4-wise group, *YASA-T₄* is faster than the other testing strategies for fourteen configurable systems. *CASA-T₄* is the fastest for eight configurable systems. *Chavatal-T₄* is the fastest for five configurable systems. *YASA-T₄* was the only strategy that was able to generate a list of configurations for CHECKSTYLE. This configurable system has 141 features, which lead to 2^{135} valid configurations.

Table 6.1: Total time in seconds spent by the target testing strategies

(a) Time spent in seconds by the 1- and 2-wise testing strategies

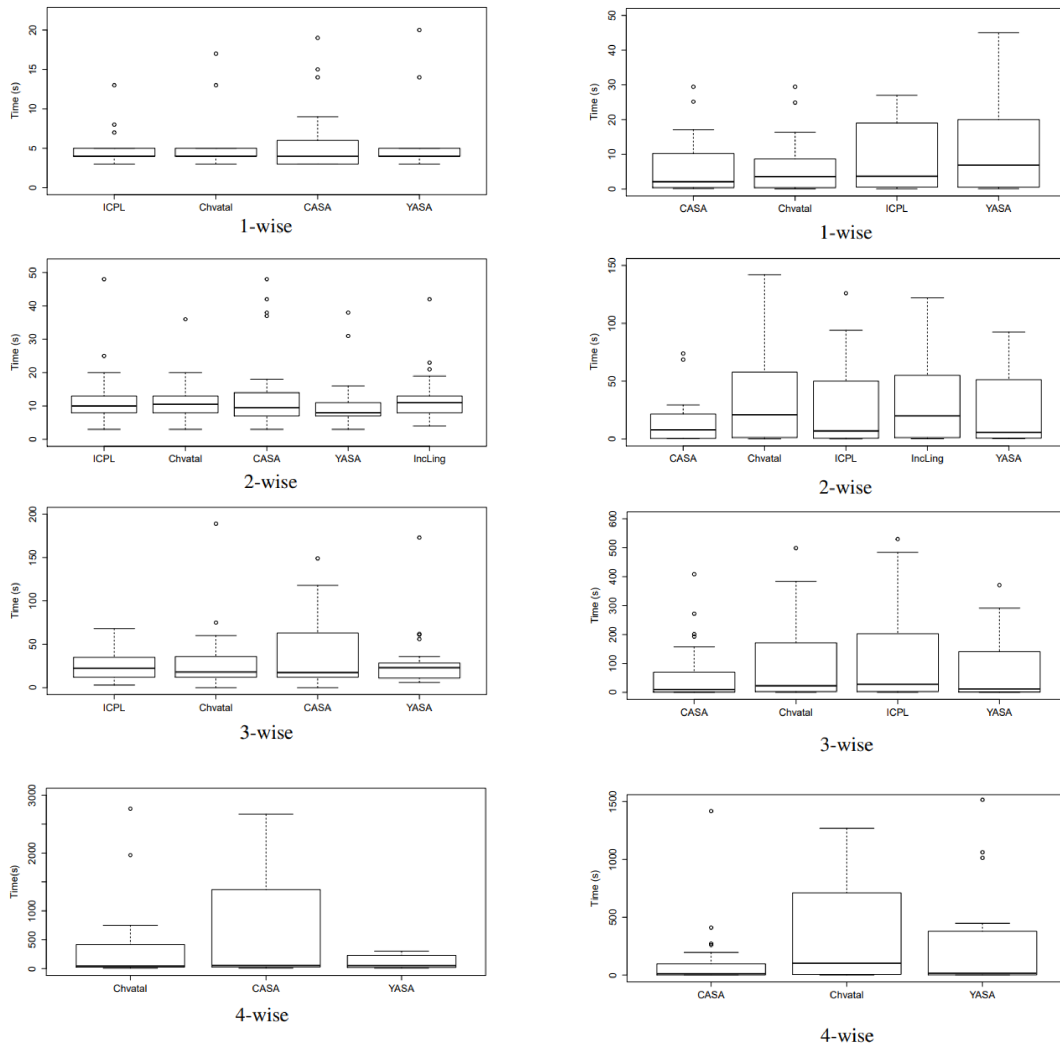
Name	Baseline1	Baseline2	CASA-T1	Chvatal-T1	ICPL-T1	YASA-T1	CASA-T2	Chvatal-T2	ICPL-T2	IncLing-T2	YASA-T2
ArgoUML-SPL	2376.00	3336.00	4.50	15.00	31.00	8.48	8.00	41.00	38.00	27.00	17.88
ATM	532.87	700.33	8.27	28.59	8.48	11.73	27.55	33.85	14.48	30.01	32.92
BankAccount	142.75	207.06	3.33	4.12	3.06	3.33	7.50	8.24	9.06	10.25	8.38
CheckStyle	1007.00	1048.00	31.16	36.00	12.00	15.26	116.65	118.00	65.00	78.00	66.93
Chess	21.23	259.23	7.92	11.17	5.61	8.88	14.65	12.13	7.61	35.00	9.87
Companies	94.41	264.41	5.44	4.51	5.52	4.03	15.14	9.26	9.52	12.30	8.86
Elevator	9.40	243.40	3.39	4.10	5.10	4.38	6.40	7.20	7.10	8.20	6.36
Email	172.00	470.00	9.69	14.00	12.00	11.68	36.25	23.00	16.00	319.00	17.00
FeatureAMP1	2410.00	2629.00	4.14	5.90	222.68	4.13	17.20	31.00	228.68	231.30	16.68
FeatureAMP2	5234.00	5426.00	14.23	28.00	25.65	11.68	21.14	92.00	32.65	213.95	76.38
FeatureAMP3	1155.00	789.00	15.65	25.00	17.92	6.97	25.65	68.00	32.92	40.00	65.67
FeatureAMP4	2718.00	2427.00	33.45	15.00	25.24	33.45	39.45	109.00	32.24	57.00	226.36
FeatureAMP5	7470.00	6809.00	15.39	66.00	59.00	20.31	23.65	155.00	68.00	133.00	104.56
FeatureAMP6	781.00	920.00	18.87	12.00	12.00	9.56	63.67	39.00	23.00	57.00	20.91
FeatureAMP7	660.00	234.30	9.32	4.50	4.70	4.29	12.34	13.70	11.70	14.00	10.37
FeatureAMP8	1998.00	2161.00	20.12	48.00	17.00	17.68	39.26	45.00	24.00	55.00	71.40
FeatureAMP9	1975.00	2534.00	6.44	23.00	24.00	27.90	82.91	87.00	33.00	103.00	66.72
GPL	41.00	102.00	7.38	5.50	5.70	5.43	14.55	16.20	13.70	18.20	12.71
IntegerSetSPL	4.00	84.00	3.36	3.10	3.10	3.34	3.41	3.10	3.10	4.20	3.40
Jtopas	22.00	91.00	3.81	3.70	3.50	4.34	6.75	7.70	7.50	6.80	7.30
MinePump	666.00	703.00	3.44	3.40	3.50	3.37	6.44	10.00	8.50	9.20	6.49
Notepad	22460.00	22557.00	201.30	299.00	235.00	197.34	450.00	1646.00	241.00	1077.00	721.92
Paycard	442.00	539.00	168.48	68.45	72.00	68.58	171.48	207.31	74.00	172.29	171.36
Prop4J	11.00	108.00	3.38	4.30	3.20	4.40	10.44	15.40	11.20	12.80	8.49
Sudoku	14.80	103.40	3.50	5.00	3.80	4.53	7.80	8.70	7.80	9.80	7.92
TaskObserver	77.00	778.00	23.80	25.00	25.00	24.80	45.50	82.70	60.00	88.00	39.72
Telecom	22.00	778.00	16.77	20.00	9.90	16.73	20.73	27.00	27.90	17.00	40.70
UnionFindSPL	10.90	103.80	9.80	11.80	10.70	10.20	19.54	12.80	8.70	17.90	11.79
VendingMachine	110.40	103.80	6.00	4.40	4.40	6.32	8.10	15.80	10.40	10.70	12.25
ZipMe	15.00	99.00	4.20	4.90	4.80	5.32	7.30	11.00	10.80	7.70	7.37

(b) Time spent in seconds by the 3- and 4-wise testing strategies

Name	Baseline1	Baseline2	CASA-T3	Chvatal-T3	ICPL-T3	YASA-T3	CASA-T4	Chvatal-T4	YASA-T4
ArgoUML-SPL	2376.00	3336.00	27.84	66.00	39.00	18.77	63.50	153.00	89.19
ATM	532.87	700.33	39.08	122.27	45.08	65.62	125.53	171.55	146.55
BankAccount	142.75	207.06	17.52	19.42	18.18	16.52	49.40	38.25	35.00
CheckStyle	1007.00	1048.00	3487.00	364.08	2869.00	368.76	**	**	18691.00
Chess	21.23	259.23	23.87	24.32	19.43	14.83	17.00	24.01	16.00
Companies	94.41	264.41	19.29	23.13	20.16	20.18	43.38	53.84	40.20
Elevator	9.40	243.40	12.41	12.30	12.30	10.48	19.54	17.50	22.49
Email	172.00	470.00	47.92	70.00	272.00	26.93	58.88	1012.00	42.28
FeatureAMP1	2410.00	2629.00	194.00	55.40	265.00	158.67	1304.00	1791.00	1226.22
FeatureAMP2	5234.00	5426.00	461.42	420.00	111.32	398.84	1235.01	1249.00	1267.60
FeatureAMP3	1155.00	789.00	214.42	283.00	97.00	175.45	1379.63	2614.00	705.00
FeatureAMP4	2718.00	2427.00	44.45	389.00	127.00	245.27	2681.39	3758.00	614.35
FeatureAMP5	7470.00	6809.00	312.91	540.00	163.00	327.04	1223.77	2222.00	1752.37
FeatureAMP6	781.00	920.00	2759.71	279.00	119.00	135.48	28863.25	6881.00	526.40
FeatureAMP7	660.00	234.30	95.41	45.20	37.70	35.42	28800.42	751.50	272.66
FeatureAMP8	1998.00	2161.00	62.87	145.00	64.00	182.69	28897.42	1105.00	584.86
FeatureAMP9	1975.00	2534.00	306.36	321.00	33.00	177.15	28963.06	1088.00	706.68
GPL	41.00	102.00	29.18	34.70	40.50	29.32	98.85	71.40	60.64
IntegerSetSPL	4.00	84.00	3.41	3.10	3.10	***	*	*	*
Jtopas	22.00	91.00	9.78	11.40	11.70	10.89	13.70	22.30	16.01
MinePump	666.00	703.00	10.55	39.00	12.60	11.53	29.38	149.00	21.81
Notepad	22460.00	22557.00	496.00	2149.00	836.00	3008.98	9782.11	5234.00	3613.55
Paycard	442.00	539.00	207.40	207.34	171.01	206.51	203.28	205.90	400.11
Prop4J	11.00	108.00	38.50	30.80	30.50	24.66	249.76	37.20	63.01
Sudoku	14.80	103.40	13.80	16.00	13.80	14.25	19.10	21.00	20.48
TaskObserver	77.00	778.00	84.21	107.00	114.00	72.13	85.65	101.00	76.16
Telecom	22.00	778.00	22.68	33.20	45.00	***	*	*	*
UnionFindSPL	10.90	103.80	12.61	12.80	12.90	14.45	17.69	14.20	14.55
VendingMachine	110.40	103.80	25.30	16.20	16.70	25.30	45.22	41.90	47.70
ZipMe	15.00	99.00	13.40	14.70	13.70	11.45	33.40	28.00	21.63

*Number of features of target configurable systems is less than t. **Could not generate configurations, time greater than 8 hours.

***The strategy did not generate any configuration.



(a) Time to generate the configurations (b) Time to run the test configurations

Figure 6.1: Time to run and to generate the list of configurations

RQ1.1. Comparing the time to generate the list of configurations. Figure 6.1a presents boxplots with the time taken to generate the list of configurations for each testing strategy organized according to the t-wise group. To improve visualization, we removed outliers. The horizontal axis presents the testing strategy, while the vertical axis presents the time spent in seconds to generate the list of configurations. For instance, Figure 6.1a *1-wise* shows the results for 1-wise group. As expected, t-wise testing strategies with smaller ‘ t ’ took less time than strategies with larger ‘ t ’. For instance, while 1-wise testing strategies took around 4 seconds, 4-wise testing strategies took around 136 seconds.

Table 6.2: P-value for RQ1 (time to generate and to execute configurations)

RQ	1-wise	2-wise	3-wise	4-wise
<i>RQ1.1</i>	0.1754	0.0013	0.3441	0.5273
<i>RQ1.2</i>	0.0795	0.0013	0.0013	0.9636
<i>RQ1</i>	0.0907	0.0003	0.9658	0.1690

In the first row of Table 6.2, we present the results of the *p-value* for the Friedman’s Test grouped according to the t-wise group. As we can see, only the time to generate the list of configurations of 2-wise testing strategies is statistically significant different from the other testing strategies (i.e., *p-value* < 0.05). It means that for 1-, 3-, and 4-wise strategies, we could not find a statistical difference on the time to generate the configurations. Therefore, we reject the null hypothesis (H0) and accept the alternative hypothesis (H1) only for the 2-wise testing strategies.

Aiming at finding out which 2-wise testing strategies differ from each other, we used the Post Hoc Analysis (see our operationalization in Section 6.4). As a result, *CASA* and *YASA* are the ones that statistically differ for the other 2-wise testing strategies. We noted that *YASA* took less time to generate the configurations for systems with more than 20 features. *CASA*, on the other hand, spent more time for systems with more than 20 features.

RQ1.2. Comparing the time to execute the test suite for the suggested configurations. Similar to Figure 6.1a, Figure 6.1b shows the boxplots with the time taken to execute the test suite for the suggested configuration by each testing strategy organized by t-wise group. In general, *CASA* took less time than the other testing strategies and *Chvatal* took more time than the other testing strategies. The greater exception was on for the 1-wise group, of which *Chvatal* was the fastest testing strategy on running the test suite.

In the second row of Table 6.2, we present the results of the *p-value* performing the Friedman’s Test. We found statistical difference for the 2- and 3-wise groups. Therefore, for these groups, we can reject the null hypothesis (H0) and accept the alternative hypothesis (H1). Regarding the Post Hoc Analysis for these two groups, we found that: (i) in the case of 2-wise testing strategies, *CASA*, *ICPL*, and *YASA* are statistically different from the other testing strategies. For the 3-wise group, *CASA* and *YASA* testing strategies are statistically different from the others. In general, *CASA* configurations were faster on running the test suite of the suggested configurations than *ICPL* and *YASA* testing strategies.

RQ1. Considering both the time to generate and to execute the suggested configurations. Finally, we performed the analysis for the sum of the time to generate and to execute the configurations. In the third row of Table 6.2, we present the results of the *p-value* for the Friedman’s Test grouped according to the t-wise group. As we can see, only the time of 2-wise testing strategies is statistically significant different from the other testing strategies (i.e., *p-value* < 0.05). As a result of the Post Hoc Analysis for 2-wise group, the time of *IncLing*, *Chvatal*, and *CASA* testing strategies are statistically different from the other testing strategies. .

Implications. Practitioners should look at our results to estimate how long their test suite might take for each testing strategy. To do so, they can compare the time that systems with similar characteristics to theirs last for each testing strategy. For instance, ARGOUML-SPL, with 153977 lines of code, 256 valid configurations, and with 1326 test cases took around 15 seconds for 1-wise strategies, 25 seconds for 2-wise strategies, 35 seconds for 3-wise strategies, and 90 seconds for 4-wise strategies

RQ₁ Summary. Although we found statistically significant results only in some cases, data of Table 6.1 show that, as expected, strategies with smaller ‘*t*’, took less time to generate and execute configurations than strategies with greater ‘*t*’. Looking the result for the same t-wise group, *ICPL-T1*, *ICPL-T2*, *ICPL-T3*, and *YASA-T4* were faster than the other testing strategies.

6.6 The Most Comprehensive Strategies (RQ₂)

Table 6.3 presents the number of valid configurations (*#Conf.*) and the percentage of configurations recommended by each testing strategy and configurable system. We highlight the greatest percentage for each configurable system. Once multiple strategies have the same greatest percentage and it is greater than 0, we consider those as the most comprehensive ones. We included baseline results in Table 6.3 aiming at fostering discussions (Section 6.10).

As a result, for 1-wise group, *Chvatal-T1* is the most comprehensive testing strategy for 15 configurable systems. *ICPL-T1*, *YASA-T1*, and *CASA-T1* are the most comprehensive testing strategies for 9, 7 and 4 configurable systems, respectively. In the 2-wise group, *IncLing-T2* is the most comprehensive testing strategy for 15 configurable systems. *ICPL-T2*, *Chvatal-T2*, *CASA-T2*, and *YASA-T2* are the most comprehensive testing strategies for 11, 9, 1 and 1 configurable systems, respectively. For 3-wise group, *Chvatal-T3* is the most comprehensive testing strategy for 20 configurable systems. Next, *ICPL-T3*, and *YASA-T3* are more comprehensive for 9 and 2

Table 6.4: P-value results for pertence of configurations analyzed

	1-wise	2-wise	3-wise	4-wise
p-value	0.9864	0.9493	0.1000	0.1292

configurable systems, respectively. *CASA-T3* did not have better results for any configurable system. For the group 4-wise, *Chvatal-T4* is the most comprehensive testing strategy for 17 configurable systems. After, *CASA-T4*, and *YASA-T4* are the most comprehensive testing strategies for 8 and 3 configurable systems, respectively.

As expected, increasing the ‘ t ’, the number of suggested configurations also increases. For systems with more than 20 features for groups 1- and 2-wise, the percentage of configurations is less than 1 % of configurations. For groups 3- and 4-wise, for configurable systems with more than 20 features, the percentage of configurations is less than 3%. For systems with 10 to 20 features for groups 1- and 2-wise, the highest percentage of configurations was for UNIONFINDSPL through *Chvatal-T2* with 60 % of configurations. For 3- and 4-wise groups with systems from 10 to 20 features, the greatest percentage was 100% for UNIONFINDSPL through *Chvatal-T3* and *Chvatal-T4*.

In three specific scenarios, we were not able to generate the configurations, for INTEGERSETSPL, 4-wise testing strategies did not work because that the number of valid configurations was smaller than 4 (see Chapter 2). For TELECOM and INTEGERSETSPL, *YASA-T3* could not generate any configuration for testing. CHECKSTYLE, *CASA-T4* and *Chvatal-T4* did not generate any configuration after eight hours and we interrupted their execution. It was not possible to reject the null hypothesis for RQ₂. Table 6.4 shows the p-values obtained running the Friedman’s Test.

Practitioners should look at our results to estimate how comprehensive the test suite of their system is for each testing strategy as well as know the constraints and limitations of testing strategies themselves. For instance, for systems with up to 8 valid configurations, most testing strategies recommend at least half of the configurations. For systems with more than 6000 configurations, testing strategies recommend no more than 2% of the valid configurations. For systems with a very small or very large number of configurations, 3- and 4-wise testing strategies may not work correctly because it did not fulfill the requirements of the t-wise strategies or did not have enough memory to run all configurations, respectively.

RQ₂ Summary. Even though the Friedman’s Test does not show that the testing strategies differ in the number of suggested configuration, our data in Table 6.3 show that *Chvatal* seems the most comprehensive testing strategy for 1-, 3-, and 4-wise testing strategies. Among 2-wise strategies, *IncLing-T2* is likely the most comprehensive testing strategy.

6.7 The Most Effective Strategies (RQ₃)

As we found faults in only 16 configurable systems, it is meaningfulness report recall for all configurable systems of the subject dataset. Hence, in this section, we focus on the 16 faulty systems. Table 6.5 presents the number of faults (*#faults*) found and the recall of each testing strategy for each configurable system with faults. To increase readability, we replace 0% to “-”. We highlight the most effective testing strategy for each configurable system (i.e., the greatest recall). Once multiple strategies have the same greatest recall and it is greater than 0%, we consider those as the most effective ones. Similar to previous sections, we present the results for baseline strategies in Table 6.5 aiming at fostering discussions in Section 6.10.

Regarding the 1-wise testing strategies, *Chvatal-T1* faced better finding more faults in 3 configurable systems and 4.25% of the total of faults in the reference list. *CASA-T1*, *YASA-T1*, and *ICPL-T1* come next finding 3.89%, 3.54%, and 3.01% of the faults. Note that for three configurable systems (*FEATUREAMPP1*, *FEATUREAMPP2*, and *MINEPUMP*) only one testing strategy suggested configurations that the test suite identified faults (*ICPL-T1* in the first configurable system and *Chvatal-T1* in the others). Regarding 2-wise testing strategies, *ICPL-T2* is the most effective testing strategy finding 15.61% of the faults. Next, *CASA-T2*, *Chvatal-T2*, *IncLing-T2*, and *YASA-T2* found 13.31%, 13.18%, 9.46%, and 5.27% of the faults from the reference list, respectively. Regarding 3- and 4-wise groups, *Chvatal-T3* and *Chvatal-T4* are the most effective testing strategy with 23.12% and 30.17% of the faults. Surprisingly, *CASA-T3* and *CASA-T4* downgraded the results from *CASA-T2* reducing the percentage of faults found (from 13.31% to 13.02% and 9.74%, respectively).

Looking at the effectiveness per configurable system, *Chvatal-T4* and *YASA-T4* achieved much greater recall than *CASA-T4*. Only for *FEATUREAMP1*, *CASA-T4* achieved a greater recall than these two other testing strategies. Note that *YASA-T4* retrieves greater recall for four subject systems (*FEATUREAMP2*, *FEATUREAMP5*, *FEATUREAMP8*, and *FEATUREAMP9*) compared to the other strategies. For the

Table 6.5: Recall of testing strategies

(a) Recall of 1- and 2-wise testing strategies

Name	#faults	Baseline1	Baseline2	CASA-T1	Chvatal-T1	ICPL-T1	YASA-T1	CASA-T2	Chvatal-T2	ICPL-T2	IncLing-T2	YASA-T2
ATM	4	100.00%	100.00%	-	-	-	-	-	-	-	-	-
BankAccount	4	100.00%	100.00%	-	-	-	-	25.00%	-	-	-	-
Chess	30	100.00%	100.00%	16.67%	16.67%	3.33%	16.67%	60.00%	30.00%	70.00%	56.67%	16.67%
Companies	17	100.00%	100.00%	11.67%	-	5.88%	-	-	29.41%	11.76%	5.88%	-
FeatureAMP1	439	18.91%	47.84%	-	-	0.23%	0.46%	-	0.46%	0.23%	0.23%	0.23%
FeatureAMP2	148	9.46%	28.38%	-	0.68%	-	-	-	1.35%	0.68%	0.68%	0.68%
FeatureAMP3	180	5.56%	29.44%	-	-	0.56%	-	-	1.11%	5.56%	0.56%	2.78%
FeatureAMP4	147	13.61%	78.91%	-	-	-	-	-	-	-	-	0.68%
FeatureAMP5	5	80.00%	-	-	-	-	-	-	-	-	-	-
FeatureAMP6	24	-	79.17%	-	-	-	-	4.17%	-	-	-	-
FeatureAMP8	4	50.00%	-	-	-	-	-	-	-	-	-	-
FeatureAMP9	236	11.44%	22.88%	0.42%	0.42%	0.42%	0.42%	2.54%	2.97%	2.97%	3.81%	2.12%
GPL	23	100.00%	100.00%	-	4.35%	4.35%	4.35%	8.70%	13.04%	26.09%	4.35%	8.70%
MinePump	24	100.00%	100.00%	-	12.05%	-	-	12.50%	12.50%	12.50%	12.50%	12.50%
Paycard	3	100.00%	100.00%	33.33%	33.33%	33.00%	33.00%	100.00%	100.00%	100.00%	67.00%	-
Sudoku	5	100.00%	100.00%	-	-	-	-	-	20.00%	20.00%	-	40.00%
All Systems	1293	61.81%	67.91%	3.89%	4.25%	3.01%	3.45%	13.31%	13.18%	15.61%	9.46%	5.27%

(b) Recall of 3- and 4-wise testing strategies

Name	#faults	Baseline1	Baseline2	CASA-T3	Chvatal-T3	ICPL-T3	YASA-T3	CASA-T4	Chvatal-T4	YASA-T4
ATM	4	100.00%	100.00%	-	-	-	-	-	-	-
BankAccount	4	100.00%	100.00%	-	25.00%	-	-	-	25.00%	25.00%
Chess	30	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	16.67%	100.00%	100.00%
Companies	17	100.00%	100.00%	17.65%	23.53%	41.18%	17.65%	29.41%	47.06%	35.29%
FeatureAMP1	439	18.91%	47.89%	3.64%	0.68%	0.23%	5.69%	17.54%	0.68%	2.96%
FeatureAMP2	48	9.46%	28.38%	3.38%	4.05%	3.38%	4.73%	12.16%	6.76%	23.65%
FeatureAMP3	180	5.56%	29.44%	5.56%	14.44%	6.11%	1.67%	5.00%	14.44%	7.22%
FeatureAMP4	147	13.61%	78.91%	-	-	-	0.68%	-	5.44%	0.68%
FeatureAMP5	5	80.00%	-	-	-	-	-	-	-	20.00%
FeatureAMP6	24	-	79.17%	4.17%	-	-	12.50%	-	-	-
FeatureAMP8	4	50.00%	-	-	-	-	25.00%	-	-	25.00%
FeatureAMP9	236	11.44%	22.88%	5.51%	4.24%	5.93%	8.90%	6.36%	7.20%	11.44%
GPL	23	100.00%	100.00%	43.48%	13.04%	13.04%	17.39%	8.70%	26.09%	17.39%
MinePump	24	100.00%	100.00%	25.00%	25.00%	25.00%	12.50%	-	50.00%	50.00%
Paycard	3	100.00%	100.00%	-	100.00%	100.00%	-	-	100.00%	-
Sudoku	5	100.00%	100.00%	-	60.00%	40.00%	40.00%	60.00%	100.00%	60.00%
All Systems	1293	61.81%	67.91%	13.02%	23.12%	20.93%	15.42%	9.74%	30.17%	23.66%

Table 6.6: P-value results for Recall of testing strategies

	1-wise	2-wise	3-wise	4-wise
p-value	0.0380	0.5846	0.0732	0.6677

configurable systems that $YASA-T_4$ achieved greater recall have more than 20 features. On the other hand, $Chvatal-T_4$ achieves greater recall for the five subject systems COMPANIES, FEATUREAMP1, FEATUREAMP3, GPL, and SUDOKU that have from 10 to 28 features. Note that for only three configurable systems (CHESS, PAYCARD, and SUDOKU) the suggested configurations of a t-wise strategy was able to find all faults in the reference list.

Practitioners should be aware that, as expected, the more configurations they test, the greater are the chances of finding faults. In addition, despite t-wise strategies suggested configurations which cover all features of a configurable system, these strategies are still far to recommend configurations that capture most of the faults.

For instance, even though *Chvatal-T4* recommends 41.25% of the valid configurations of ATM, these configurations find no fault. Researchers may see it as an opportunity to propose sampling strategies that somehow recommend fault-prone configurations.

Table 6.6 presents the *p-values* for the Friedman’s Test according to the t-wise group. As we can see, only 1-wise testing strategies is statistically significant different from the other testing strategies (i.e., *p-value* < 0.05) on being effective on finding faults. It means that for 2-, 3-, and 4-wise strategies, there is no statistical difference on the time to generate the configurations. Therefore, we reject the null hypothesis (H0) and accept the alternative hypothesis (H1) only for the 1-wise testing strategies. With the Post Hoc Analysis for the 1-wise group, we did not obtain a significant difference among the strategies. It might be because the p-value is close to our confidence level.

RQ₃ Summary. Although we found statistically significant results only for 1-wise strategies, data of Table 6.5 show that *Chvatal-T4*, *YASA-T4*, and *Chvatal-T3* seem to be the testing strategies that suggested configurations able to find fault mostly effective. Nevertheless, there are still space for improvements since the strategy that faced better found 30.17% of the faults in the reference list.

6.8 The Most Time-Efficient Strategy (RQ₄)

Table 6.7 presents the *time-efficiency* of each testing strategy for each configurable system with faults. We use “-” to represent testing strategies of which no fault was found. We highlight the most time-efficient testing strategy for each configurable system. Once multiple strategies have the same greatest *time-efficiency*, we consider those as the most *time-efficiency* ones. Similar to previous sections, we present baseline results for fostering discussions (Section 6.10).

As a result, *Chvatal-T1* is the most *time-efficiency* testing strategy in the 1-wise group for 4 configurable systems. *YASA-T1*, *ICPL-T1*, and *CASA-T1* are the most time-efficient testing strategy for 3, 2 and 1 configurable systems, respectively. For 2-wise group, *ICPL-T2* is the most time-efficient testing strategy for 6 configurable systems. *CASA-T2*, *Chvatal-T2*, and *YASA-T2* are the most time-efficient testing strategy for 3, 2 and 2 configurable systems, respectively. *IncLing-T2* was the most time efficient testing strategy for no system. *ICPL-T3* is the most time-efficient testing strategy to the 3-wise group for 5 configurable systems. *YASA-T3*, *Chvatal-T3*, and *CASA-T3* are the most time-efficient testing strategy for 4, 2 and 2 configurable systems, respectively. Finally, in the 4-wise group, *YASA-T4* was the far most time-efficient testing strategy compared to the other testing strategies. While it was the

Table 6.7: Time Efficiency of testing strategies

(a) Time Efficiency of 1- and 2-wise testing strategies

Name	Baseline1	Baseline2	CASA-T1	Chvatal-T1	ICPL-T1	YASA-T1	CASA-T2	Chvatal-T2	ICPL-T2	IncLing-T2	YASA-T2
ATM	0.008	0.006	-	-	-	-	-	-	-	-	-
BankAccount	0.028	0.019	-	-	-	-	0.133	-	-	-	-
Chess	1.413	0.116	0.631	0.448	0.178	0.563	1.229	0.742	2.760	0.631	0.507
Companies	0.180	0.064	0.368	-	0.181	-	-	0.540	0.210	0.368	-
FeatureAMP1	0.034	0.080	-	-	0.004	0.484	-	0.065	0.004	0.004	0.060
FeatureAMP2	0.005	0.008	-	0.036	-	-	-	0.022	0.031	0.005	0.013
FeatureAMP3	0.009	0.067	-	-	0.056	-	-	0.029	0.304	0.025	0.076
FeatureAMP4	0.024	0.048	-	-	-	-	-	-	-	-	0.004
FeatureAMP5	0.001	-	-	-	-	-	-	-	-	-	-
FeatureAMP6	0.026	0.021	-	-	-	-	0.016	-	-	-	-
FeatureAMP8	0.001	-	-	-	-	-	-	-	-	-	-
FeatureAMP9	0.024	0.021	0.155	0.043	0.042	0.036	0.072	0.080	0.212	0.155	0.075
GPL	0.561	0.575	-	0.182	1.428	0.184	0.137	0.185	0.438	0.055	0.157
MinePump	0.036	0.039	-	0.882	-	-	0.466	0.300	0.353	0.326	0.462
Paycard	0.007	0.006	0.006	0.015	0.014	0.015	0.017	0.014	0.041	0.006	-
Sudoku	0.338	0.048	-	-	-	-	-	0.115	0.128	-	0.253

(b) Time Efficiency of 3- and 4-wise testing strategies

Name	Baseline1	Baseline2	CASA-T3	Chvatal-T3	ICPL-T3	YASA-T3	CASA-T4	Chvatal-T4	YASA-T4
ATM	0.008	0.006	-	-	-	-	-	-	-
BankAccount	0.028	0.019	-	0.051	-	-	-	0.026	0.029
Chess	1.413	0.116	1.257	1.234	1.544	2.023	0.294	1.249	1.875
Companies	0.180	0.064	0.156	0.173	0.347	0.149	0.116	0.149	0.149
FeatureAMP1	0.034	0.080	0.082	0.054	0.024	0.158	0.059	0.002	0.011
FeatureAMP2	0.005	0.008	0.011	0.014	0.045	0.018	0.015	0.008	0.028
FeatureAMP3	0.009	0.067	0.047	0.092	0.113	0.017	0.007	0.010	0.018
FeatureAMP4	0.024	0.048	-	-	-	0.004	-	0.002	0.002
FeatureAMP5	0.001	-	-	-	-	-	-	-	0.001
FeatureAMP6	0.026	0.021	0.001	-	-	0.022	-	-	-
FeatureAMP8	0.001	-	-	-	-	0.006	-	-	0.002
FeatureAMP9	0.024	0.021	0.042	0.031	0.424	0.006	0.001	0.016	0.038
GPL	0.561	0.575	0.343	0.086	0.074	0.136	0.020	0.084	0.066
MinePump	0.036	0.039	0.569	0.154	0.476	0.260	-	0.081	0.550
Paycard	0.007	0.006	-	0.014	0.018	-	-	0.015	-
Sudoku	0.338	0.048	-	0.188	0.154	0.140	0.157	0.238	0.146

Table 6.8: P-value results for Time Efficiency by the strategies

	1-wise	2-wise	3-wise	4-wise
p-value	0.2939	0.9716	0.7581	0.6065

most time-efficient for 10 configurable systems, *Chvatal-T4* and *CASA-T4* were the most time-efficient testing strategies for only 4 and 1 configurable systems, respectively. Regarding the Friedman's Test, it was not possible to reject the null hypothesis. Table 6.8 shows the *p-values* obtained with the Friedman's Test.

RQ₄ Summary. Although the Friedman’s Test does not show statistically difference on the time-efficiency among testing strategies from the same t-wise group, our data in Table 4 show that *Chvatal-T1*, *ICPL-T2*, *ICPL-T3*, and *YASA-T4* are probably the testing strategies that suggested configurations able to find the best balance among the number of faults and time for the 1-, 2-, 3-, and 4-wise groups, respectively.

6.9 The Most Coverage-Efficient Strategy (RQ₅)

Table 6.9 presents the *coverage-efficiency* of each testing strategy for each configurable system with faults. We use "-" for representing testing strategies that no fault was found. We highlight the most coverage-efficient testing strategy for each configurable system. Once multiple strategies have the same greatest coverage-efficiency, we consider those as the most coverage-efficient ones. Similar to previous sections, we present results for baseline testing strategies aiming at fostering discussions (Section 6.10).

As a result for the 1-wise group, *Chvatal-T1* is the most coverage efficient testing strategy for 3 configurable systems. *CASA-T1*, *YASA-T1*, and *ICPL-T1* are the most coverage-efficient testing strategies for 2, 2, and 1 configurable systems, respectively. For the 2-wise group, *CASA-T2* and *Chvatal-T2* are the most coverage-efficient testing strategy for 5 configurable systems. *ICPL-T2*, *IncLing-T2*, and *YASA-T2* are the most coverage-efficient testing strategies for 2, 2, and 1 configurable systems, respectively. For the 3-wise group, *Chvatal-T3* is the most coverage-efficient testing strategy for 5 configurable systems. *ICPL-T3*, *YASA-T3*, and *CASA-T3* are the most coverage-efficient testing strategies for 3, 3, and 2 configurable systems, respectively. For the 4-wise group, *Chvatal-T4* is the most coverage-efficient testing strategy for 6 configurable systems. *CASA-T4* and *YASA-T4* are the most coverage-efficient testing strategies for 2 configurable systems, respectively.

Note that since each t-wise testing strategy faced better in at least one configuration system, the *coverage-efficiency* analysis presented more dispersed results than the previous ones. In Table 6.10, we show the *p-values* obtained with the Friedman’s Test. As seen, it was not possible to reject the null hypothesis in any case.

RQ₅ Summary. Although the Friedman’s Test does not show statistically difference on the coverage-efficiency among testing strategies from the same t-wise group, our data of Table 6.9 show that *Chvatal* is the testing strategy that recommends configurations able to find the best balance among faults and the number of recommended configurations in all groups (1-, 2-, 3-, and 4-wise).

Table 6.9: Coverage Efficiency of testing strategies

(a) Coverage Efficiency of 1- and 2-wise testing strategies

Name	Baseline1	Baseline2	CASA-T1	Chvatal-T1	ICPL-T1	YASA-T1	CASA-T2	Chvatal-T2	ICPL-T2	IncLing-T2	YASA-T2
ATM	0.050	0.050	-	-	-	-	-	-	-	-	-
BankAccount	0.028	0.028	-	-	-	-	0.143	-	-	-	-
Chess	4.286	4.286	2.500	2.500	0.500	2.500	4.500	3.000	4.200	4.250	2.500
Companies	0.089	0.089	0.500	-	0.250	-	-	0.385	0.154	0.077	-
FeatureAMP1	0.332	1.240	-	-	0.333	1.000	-	0.182	0.100	0.071	0.100
FeatureAMP2	0.112	0.276	-	0.333	-	-	-	0.167	0.091	0.077	0.091
FeatureAMP3	0.040	0.316	-	-	0.333	-	-	0.133	0.625	0.048	0.556
FeatureAMP4	0.080	0.516	-	-	-	-	-	-	-	-	0.100
FeatureAMP5	0.016	-	-	-	-	-	-	-	-	-	-
FeatureAMP6	-	0.080	-	-	-	-	0.071	-	-	-	-
FeatureAMP8	0.008	-	-	-	-	-	-	-	-	-	-
FeatureAMP9	0.188	0.296	0.500	0.333	0.333	0.500	0.667	0.583	0.636	0.750	0.500
GPL	0.258	0.258	-	0.250	0.200	0.167	0.143	0.231	0.120	0.071	0.118
MinePump	0.375	0.375	-	1.000	-	-	0.429	0.429	0.375	0.429	0.500
Paycard	0.500	0.500	0.500	0.500	0.500	0.500	0.600	0.500	0.600	0.400	-
Sudoku	0.250	0.250	-	-	-	-	-	0.143	0.125	-	0.286

(b) Coverage Efficiency of 3- and 4-wise testing strategies

Name	Baseline1	Baseline2	CASA-T3	Chvatal-T3	ICPL-T3	YASA-T3	CASA-T4	Chvatal-T4	YASA-T4
ATM	0.050	0.050	-	-	-	-	-	-	-
BankAccount	0.028	0.028	-	0.048	-	-	-	0.024	0.024
Chess	4.286	4.286	3.750	4.286	4.286	3.750	0.625	4.286	3.750
Companies	0.089	0.089	0.125	0.125	0.219	0.136	0.083	0.104	0.133
FeatureAMP1	0.332	1.240	0.667	0.094	0.031	0.833	1.167	0.036	0.160
FeatureAMP2	0.112	0.276	0.192	0.158	0.143	0.212	0.692	0.106	0.407
FeatureAMP3	0.040	0.316	0.250	0.500	0.204	0.064	0.136	0.228	0.096
FeatureAMP4	0.080	0.516	-	-	-	0.032	-	0.088	0.012
FeatureAMP5	0.016	-	-	-	-	-	-	-	0.011
FeatureAMP6	-	0.080	0.018	-	-	0.045	-	-	-
FeatureAMP8	0.008	-	-	-	-	0.033	-	-	0.011
FeatureAMP9	0.188	0.296	0.448	0.294	0.424	0.677	0.517	0.198	0.342
GPL	0.258	0.258	0.250	0.086	0.086	0.091	0.063	0.095	0.065
MinePump	0.375	0.375	0.400	0.286	0.316	0.231	-	0.333	0.444
Paycard	0.500	0.500	-	0.500	0.500	-	-	0.500	-
Sudoku	0.250	0.250	-	0.200	0.143	0.143	0.214	0.250	0.150

Table 6.10: P-value results for Coverage Efficiency of testing strategies

	1-wise	2-wise	3-wise	4-wise
p-value	0.1893	0.8847	0.7961	0.5258

6.10 Grouping Testing Strategies

While Section 6.5 to Section 6.9 present broader results, this section presents results grouping similar strategies.

1-Wise Strategies. Comparing *ICPL-T1* and *Chvatal-T1*, we see that both are similarly fast, but little comprehensive, very little effective, and little time efficient. *Chvatal-T1* seems a little bit more time efficient than *ICPL-T1*.

2-Wise Strategies. Comparing *ICPL-T2*, *Chvatal-T2*, and *IncLing-T2*, we see that *ICPL-T2* is faster than the others, specially in the case of FEATUREAMP9. *Chvatal-T2* and *IncLing-T2* are more comprehensive than *ICPL-T2*. *Chvatal-T2* has a greater recall than the others. *IncLing-T2* is more time and coverage efficient than the others.

3-Wise Strategies. Comparing *ICPL-T3* and *Chvatal-T3*, we see that they are similarly slow and comprehensive. Nevertheless, *Chvatal-T3* is slightly more effective and time- and coverage-efficient than *ICPL-T3*.

ICPL Strategies. *ICPL-T1* is normally faster and slightly coverage efficient than *ICPL-T2* and *ICPL-T3*. *ICPL-T3* more comprehensive than the others. *ICPL-T2* and *ICPL-T3* are similarly more effective than *ICPL-T1*. *ICPL-T2* is more time efficient than the other strategies.

Chvatal Strategies. *Chvatal-T1* is faster and slightly more time efficient than the others. *Chvatal-T4* is more comprehensive and effective than the others. *Chvatal-T3* and *Chvatal-T4* are more coverage efficient than the others.

T-wise versus Baselines. We started using the baselines to increase the number of configurations tested. However, our baselines remembered us that, looking at a minimized number of configurations leaves a bunch of configurations behind where unexpected feature interactions faults may occur. For instance, while baseline 1 and 2 found 856 and 795 faults, *Chvatal-T4* found only 353 faults, which is less than half of the faults found by the baselines. T-wise strategies follow an algorithm to select configurations that cover a combination of all features (see Chapter 2). On the other hand, *baseline1* simply looks at a predefined number of configurations sequentially (e.g., brute force) and *baseline2* randomly chooses a predefined number of configurations. Hence, we do not question the logic behind, we only investigate the time, coverage, effectiveness, time efficiency, and coverage efficiency of the configurations recommended by all testing strategies investigated in this study. As a result, t-wise strategies are faster, more time and coverage efficient than our baselines. On the other hand, the baselines are more comprehensive and effective than t-wise strategies.

All these comparisons agree with our results in Section 6.6 by showing that more comprehensive strategies (i.e., the recommend a larger number of configurations) often have greater effectiveness. However, testing more configurations makes the testing time-consuming. Regarding time and coverage efficiency, it depends on both the number of configurations and faults present in the target configurable system.

6.11 Implications for Practitioners

After reading this chapter a question may arise:

Which testing strategy should I use?

The answer varies depending on your *system*, *project life-cycle*, and *organizational constraints*. In the ideal case, you should test all configurations. In practice, the maximal number of feasible configurations, as explained next. Looking at the results of Sections 6.5 and 6.6, practitioners have a practical estimation of time and coverage of each testing strategy. For instance, for small projects, such as GPL (1235 lines of code, 13 features, and 51 test cases), it is reasonable waiting around 10 milliseconds to run the testing suite of all 73 valid configurations. On the other hand, for large projects such as CHECKSTYLE (61435 lines of code, 141 features, and 719 test cases), it is not feasible or meaningful wait until finishing running the test suite of all $>2^{135}$ configurations. Our suggestion is to first run configurations that test the source code of changed features. Then, choose a t-wise strategy that fits your time- and coverage-constraints to have a general view of the project.

Aiming at supporting the choice of a testing strategy, we condensed our results in Figure 6.2 representing each t-wise group separately, only t-wise strategies, and all testing strategies used in this study (i.e., including the baselines). For example, in Figure 6.2a, we show the 1-wise testing strategies, in Figure 6.2b, we show the 2-wise group, in Figure 6.2e, we show the comparison between t-wise groups, and, in Figure 6.2f, we show all testing strategies used in this study. The broader the line, the better the strategy is for a target characteristic. Next, we discuss each sub-figure individually.

1-Wise testing strategies (Figure 6.2a). Comparing the 1-wise strategies, *ICPL-T1* stood out as the fastest strategy. *Chvatal-T1* was the strategy with the greatest coverage and recall, and was the most time- and coverage-efficient.

2-Wise testing strategies (Figure 6.2b). Comparing the 2-wise strategies, *ICPL-T2* was fastest and time-efficient and got the greatest coverage and recall. *CASA-T2* and *Chvatal-T2* were the most coverage-efficient strategies.

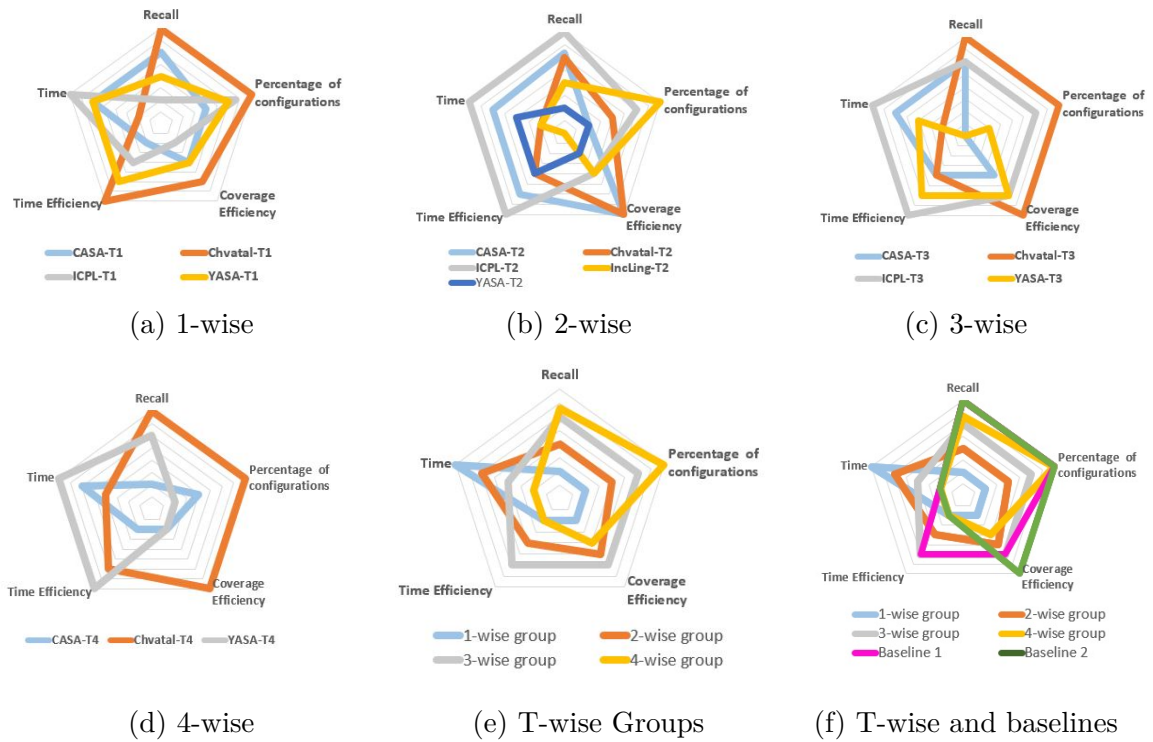


Figure 6.2: Summary of t-wise strategies comparison

3-Wise testing strategies (Figure 6.2c). Comparing the 3-wise strategies, *ICPL-T3* was fastest and time-efficient testing strategy. *Chvatal-T3* achieved better coverage, recall, and coverage-efficient.

4-Wise testing strategies (Figure 6.2d). Comparing the 4-wise strategies, *YASA-T4* was the fastest and time-efficient testing strategy. *Chvatal-T4* covered more configurations, achieved the greatest recall and was the most coverage-efficient testing strategy.

T-wise testing strategies (Figure 6.2e). If there is a great time-constraint, 1-wise strategies might be the right option. 2-wise strategies found faults that no 1-wise strategy found in *BANKACCOUNT*, *FEATUREAMP4*, *FEATUREAMP6*, and *SUDOKU*. For practitioners who want to test more configurations than the suggested by 1-wise strategies and still has a great time-constraint, 2-wise strategies is a good option. 3-wise strategies obtained greater time-efficiency and coverage-efficiency results than the other groups. Note that 3-wise strategies recall was close to the recall of 4-wise strategies. Practitioners might choose 3-wise group when prioritizing time- and coverage-efficiency. When generating configurations using 4-wise strategies we noticed that they took a while for systems with more than 20 features. However, considering that practitioners do not need to generate the configurations every time (e.g., only when the feature module changes), they can reuse the suggested list of configurations multiple times.

This point make 4-wise strategies more usable in practice.

Comparing all testing strategies in the study (Figure 6.2f). As expected, the baselines used in the study got better results for recall, percentage of configurations, and coverage efficiency than the t-wise strategies. It happened because our baselines test all possible configurations for systems with less than 250 valid configurations. When the total number of valid configurations is small and there is not a strong time-constraint, testing all configurations is the best option. However, for systems with a much greater number of valid configurations, it is infeasible in practice.

We started using the baselines to increase the number of configurations tested. However, our baselines reminded us that, looking at a minimized number of configurations leaves a bunch of configurations behind where feature interactions faults may occur. For instance, while baseline 1 and 2 found 856 and 795 faults in the subject dataset, *Chvatal-T4* found only 353 faults. It is less than half of the faults found by the baselines. Hence, depending on the number of valid configurations, it is meaningful to test all of them.

6.12 Implications for Researchers and Tool Builders

Our study does not aim to point out which testing strategy has a better performance on finding configurations that will be recommended. We only use the testing strategies' outcome (i.e., their suggested configurations) to run the respective testing suite for the recommended configurations and report results from the subject dataset. Therefore, we are not comparing the testing strategies themselves but their recommended configurations.

Our effectiveness results (Section 6.7) show that even when using sophisticated t-wise testing strategies, it is difficult to find faults with the generated configurations. In the best case using a t-wise strategy, we retrieve 30.62% of faults. We see two prominent directions to improve testing strategies and to increase the chance of finding more feature interaction faults. The first uses Machine Learning techniques to retrieve information of faulty-features and faulty-classes. Hence, this information can be used as an additional source for t-wise strategies. The second regards the creation of a strategy that interactively asks developers for components that they want to prioritize. Hence, the strategy should generate a set of configurations that exhaustively investigate the target components.

Both researchers and tool builders benefit from these directions. While researchers have the opportunity to propose testing strategies using information not yet

used for this purpose, tool builders can automate new strategies leaving them ready for practitioners to use.

6.13 Threats to Validity

Even with a careful planning, this research can be affected by different factors which might threaten our findings [Wohlin, 2014]. We discuss these factors and decisions to mitigate their impact on our study divided into external and internal threats to validity below.

External Validity. External validity is threatened mainly by two factors. First, our restriction to variability encoding as variability approach and JAVA as programming language. The generalization to other variability approaches, a programming languages, and configurable systems is limited. This limitation of the sample was necessary to reduce the influence of confounds, increasing internal validity, though [Siegmond and Schumann, 2015]. While more research is needed to generalize to other variability approaches, programming languages, and configurable systems, we are confident that we selected and analyzed a practically relevant variability approach and a substantial number of configurable systems from various domains, longevity, size, and valid configurations. Limiting the programming language is a common limitation of several research studies on configurable software systems [Kim et al., 2012b, 2013; Meinicke et al., 2014; Souto et al., 2017; Wong et al., 2018]. Second, the testing suite and faults found in the dataset. Our results are restricted to the test suite and faults found in the subject dataset. Using other systems, different versions of the subject systems, or different testing suites may come up with other results. Aiming at minimizing this threat, we chose a dataset previously proposed 4 that follows two reasonable thresholds to increase the quality of the testing suite: 70% of code coverage and 40% of killed mutants.

Internal Validity. There are three major threats to the internal validity of our study. First, we could have wrongly implemented the subject testing strategies. However, as we use the implementation provided on FEATUREIDE, we are confident that they were correctly implemented. Second, we cannot ensure that we identified all faults present in the subject systems. To increase testing coverage, we use two baselines. While the first baseline runs the test suite in all valid configurations for up to 250 valid configurations sequentially chosen, the second baseline runs the testing suite for up to 250 valid configurations randomly chosen. This way we run all valid configurations for several subject systems. In addition, our reference list is the union set of

all faults found by all configurations run by the ten subject testing strategies. Hence, we are not prioritizing faults found by one testing strategy, and all testing strategies have the same chance of finding faults.

Third, we may have not chosen the best metrics to represent systems, features, and classes. We selected several well-known metrics to quantify the size, features, classes, and test suite that compose the configurable systems of the subject dataset. To make this measurement process simpler and automated, we used well-known tools, such as CK TOOL [CK, 2020] and JACOCo [JaCoCo, 2020] and, for the metrics that well-known tools are not able to compute, we create a script. We manually checked the measurement of 5% of the components (e.g., classes, features) to confirm their results. Hence, we believe that similar conclusions would also be achieved using different metrics and tools that quantify similar attributes for the same set of configurable systems and components.

6.14 Final Remarks

In this chapter, we compared the performance of the configurations recommended by eight t-wise strategies. In this comparison, we used our dataset with 30 configurable systems and with a test suite available for each of them. Regarding the performance of a t-wise strategy, we identified which testing strategies provide configurations that were often faster, more comprehensive, more effective, more time-efficient, and more coverage-efficient. As a result, we found that: (i) *ICPL-T1* is usually fast and slightly more coverage efficient than the other strategies; (ii) *Chvatal-T4* is by far the most comprehensive testing strategy; (iii) *Chvatal-T4* is the testing strategy that recommends configurations able to find the greatest number of faults. (iv) *Chvatal-T1* is the testing strategy that recommends configurations able to find the best balance among faults found and time, and (v) *Chvatal-T4* is the testing strategy that recommends configurations able to find the best balance among faults found and the number of recommended configurations.

Our results can be used by practitioners to support their decision of which testing strategy to use. Researchers and tool builders may also benefit from our study since we provide directions for improving existing testing strategies. These directions include, for instance, using source code metrics to identify fault-prone components and use this information on existing testing strategies. Moreover, using a variant t on the testing strategy may depend on the number of valid configurations or constraints provided by practitioners when using the tool. In the next chapter, we investigate the dispersion of faults over classes and features from the dataset.

Chapter 7

Investigating the Dispersion of Faults over Classes and Features

Faults may occur in different modules of configurable software systems. It is important to know the location of faults to ensure that all configurations are correctly compiled, built, and run, developers spend considerable effort testing their systems. This effort is necessary mainly because configurations that fail may hurt potential users and degrade the reputation of a project. This chapter investigates the dispersion of faults over classes and features in configurable software systems. Aiming to understand faults' location investigated, we investigate if it is possible to distinguish failing classes and failing features from classes and features safe of faults. A deep understanding of faults may help practitioners learn the characteristics of classes and features prone to fail, avoid introducing similar faults, and guide them to increase the test coverage in these fault-prone classes and features.

We compared testing strategies in previous chapters. In this chapter, we investigated features of classes and features that failed found in the study of Chapter 6. We found at least one fault in 16 out of 30 systems in the subject dataset. Regarding the class-level analysis, we found that faults are concentrated in only 0.8% of classes of the subject dataset and high values of some source code metrics, such as *LoC*, *WMC*, and *RFC*, appear to be related to fault-prone classes. Finally, regarding the feature-level analysis, we found that faults are concentrated in a few features with high values of some source code metrics, such as feature scattering and tangling.

The remainder of this chapter is organized as follows. Section 7.1 presents our goal and research questions. Section 7.2 describes how we acquire data. Section 7.3 describe how we operationalize the answer of our research questions. Sections 7.4 and 7.5 present the results achieved regarding dispersion of faults over classes and features, respectively. Section 7.6 presents discussions of the on the relation between faulty

classes and faulty features. Section 7.7 describes the main limitations and threats to validity of this work. Finally, Section 7.8 concludes this chapter.

7.1 Goal and Research Questions

Based on the goal question metric (GQM) template [Basili and Rombach, 1988], we systematically defined our goal as follows. We *analyze* the dispersion of faults over systems, classes and features; *for the purpose* of deeply understanding the location of faults; *with respect* to support practitioners and researchers to broaden the understanding of the dispersion of faults over classes and features in configurable systems; *from the viewpoint* of researchers and software developers with expertise in software testing *in the context* of a previously proposed dataset of configurable software systems with test suite available.

To build a reference list of faults, we use the union of all faults found by the subject strategies of Chapter 6. Then, we measure each class and feature with metrics commonly used in practice [Chidamber et al., 1998]. Number of lines of code (*LoC*) [CK, 2020], weighted methods per class (*WMC*) [CK, 2020], and response for a class (*RFC*) [CK, 2020] are examples of metrics at class-level. Feature scattering and feature tangling are examples of metrics at feature-level. Finally, we compute Spearman’s rank correlation between the number of faults in a component (i.e., class or feature) and a given metric. A great understanding of these faults may benefit both practitioners and researchers because they can (i) learn patterns from previous faults, (ii) use these patterns to avoid the emergence of similar faults, and (iii) improve existing testing strategies. Aiming at investigating the dispersion of faults over classes and features, we formulate the following two research questions.

RQ₁: What are the characteristics of fault-prone classes of the configurable systems?

RQ₂: What are the characteristics of fault-prone features of the configurable systems?

7.2 Data Acquisition

Our data acquisition consists basically of three tasks: (i) run the testing strategies presented in Chapter 6 for each configurable system in the subject dataset (Chapter 4), (ii) extract information from logs creating a set of true faults (reference list), and (iii) collect metrics. In what follows, we give details on how we automated these tasks¹.

¹<https://github.com/fischerJF/Community-wide-Dataset-of-Configurable-Systems/tree/master/Tools>

Creating a Reference List. We analyzed the execution log of the 18 (16 t-wise strategies and two baselines) testing strategies against the 30 subject systems. After parsing the log, we found test cases of which a fault emerged as well as the reason why it happened. Hence, we investigated each reported fault to confirm that it is truly a fault and if it arose due to a feature interaction. The reference list is the union of all faults found on all configurations suggested by all testing strategies investigated in this study.

Metrics Collection. We use different tools to extract metrics used in our study. Once we did not find tools to compute metrics used to answer our research questions, we computed them with our scripts (see Chapter 4). Our analysis scripts (written in Java) are open-source. All data necessary for replicating this study are stored in csv files. All tools, links to subject projects, reports of faults for each testing strategy, and data used in this study are available at our supplementary website [Ferreira et al., 2020c].

7.3 Operationalization

Answering RQ₁. To answer RQ₁, we first retrieve a list of all classes that failed and discuss their dispersion over each subject configurable system. Then, aiming at discovering whether these faulty classes have distinct characteristics from other classes, we compute traditional and CK metrics [Chidamber et al., 1998] for each class of each subject system (see Section 4). After, we compute the Spearman’s rank correlation to see whether faulty classes are often larger and more complex than non-faulty classes. Spearman’s rank correlation is adequate to this analysis since the measures of our classes represent continuous values and do not follow a normal distribution.

Answering RQ₂. To answer RQ₂, we did a similar analysis to answer RQ₁. The main difference is that instead of investigating characteristics of faulty classes, we investigate characteristics of faulty features. Naturally, we use metrics related to features, such as the number of classes and methods a feature is located (scattering). Spearman’s rank correlation is adequate to this analysis since the measures of our features represent continuous values and do not follow a normal distribution. We detail the used metrics when answering this research question.

Table 7.1: Faults found by class with traditional metrics

System Name	Class Name	CBO	WMC	DIT	RFC	LCOM	NOM	NOPM	NOSM	NOF	NOFP	NOSF	NOSI	LoC	#F
ATM	ATMUserInterface	12	62	6	53	0	34	23	0	43	17	19	10	447	3
	Withdrawal	6	31	2	11	0	5	4	0	4	0	1	0	159	1
Bankaccount	Transaction	1	19	1	6	3	3	1	1	1	0	0	1	38	4
Chess	Model	6	75	1	48	76	20	16	1	18	1	1	0	289	23
	Pawn	1	24	2	9	1	2	2	0	0	0	0	0	69	1
	Board	13	196	1	47	124	40	27	0	14	0	0	0	596	6
Companies	Controller	12	61	1	18	0	7	7	0	2	0	0	0	181	17
FeatureAMP1	PlaylistHandler	4	56	1	29	13	15	10	0	9	0	0	1	158	33
	App	7	159	1	124	848	48	20	0	43	0	0	25	677	406
FeatureAMP2	Gui	11	310	7	178	3137	66	26	0	44	6	1	10	1161	148
FeatureAMP3	FeatureAmp	20	470	6	251	6511	117	17	1	57	0	5	12	1537	180
FeatureAMP4	FeatureAmp	13	78	1	56	404	33	10	3	12	1	1	8	285	93
	PlayerBar	11	16	1	17	1	10	9	0	5	0	0	0	108	55
FeatureAMP5	Main	9	211	6	159	2889	60	3	2	34	0	4	22	860	5
FeatureAMP6	Playlist	1	62	1	23	116	24	18	0	4	2	2	0	185	10
	Kernel	7	114	1	71	940	52	29	0	14	0	0	9	411	14
FeatureAMP8	Application	8	355	2	173	3987	75	39	1	37	1	0	11	1565	4
FeatureAMP9	Gui	7	311	6	176	2467	61	22	0	40	3	1	13	1172	236
GPL	Graph	10	175	1	48	241	26	25	4	3	3	1	27	797	23
MinePump	PL_Interface	3	31	1	16	34	9	8	4	5	2	5	7	186	24
PayCard	PayCard	2	31	1	6	25	10	7	1	5	0	0	0	127	3
Sudoku	BoardManager	7	99	1	33	54	21	15	0	4	0	0	12	295	5

CBO: Coupling between Objects; **WMC:** Weight Method Class; **DIT:** Depth Inheritance Tree; **RFC:** Response for a Class; **LCOM:** Lack of Cohesion of Methods; **NOM:** Number of Methods; **NOPM:** Number of Public Methods; **NOSM:** Number of Static Methods; **NOF:** Number of Fields; **NOFP:** Number of Public Fields; **NOSF:** Number of Static Fields; **NOSI:** Number of Static Invocations; **LoC:** Lines of Code; **#F:** Number of Faults Found.

7.4 Dispersion of Faults over Classes (RQ₁)

Table 7.1 reports the 22 classes with faults distributed over 16 configurable systems. For each faulty class, we present traditional source code metrics, such as Coupling between Object Classes (CBO), Weighted Methods per Class (WMC), Depth Inheritance Tree (DIT), Number of Methods (NOM), Lines of Code (LoC), and Faults Found (#F). We analyze the log generated by JUNIT to understand each fault found. Through the JUNIT log, we evaluated each faulty class. Table 7.2 presents the 22 classes with faults concerning feature metrics, such as Total number Features that handle the related class (TNF), Scattering over classes (ScC), and Tangling in class (TaC).

Considering that the subject dataset has 2740 classes and we found faults in 22, only 0.8% of the classes failed. It shows that faults are highly concentrated in a few classes. By looking only at the 16 faulty systems, we see that in 11 systems the faults are concentrated in only one class and in the other 5 systems (ATM, CHESS, FEATUREAMP1, FEATUREAMP4, and FEATUREAMP6), the faults are located in up to three classes.

Looking at the number of faults per faulty class, we see that most of the faults found are in few classes. For instance, 31.40% of the faults are in the class *App* of FEATUREAMP1, 18.35% of the faults are in the class *Gui* of FEATUTUREAMP9, and

7. INVESTIGATING THE DISPERSION OF FAULTS OVER CLASSES AND FEATURES 92

Table 7.2: Faults found by class with variability metrics

System	Class Name	TNF	ScC	VarC	TaC	MFA	LoCIF	#F
ATM	ATMUserInterface	6	10	32	15	2	40	3
	Withdrawal	2	4	3	7	3	59	1
bankaccount	Transaction	1	1	1	1	1	1	4
chess	Model	0	0	0	0	0	0	1
	Board	3	10	20	6	8	22	23
	Pawn	0	0	0	0	0	0	6
Companies	Controller	5	119	155	85	3	16	17
FeatureAMP1	PlaylistHandler	3	8	11	7	5	10	33
	APP	17	23	30	22	24	51	406
FeatureAMP2	GUI	19	35	55	29	37	85	148
FeatureAMP3	FeatureAmp	26	47	90	39	58	163	180
FeatureAMP4	FeatureAmp	12	50	35	24	2	61	93
	PlayerBar	1	9	3	2	1	28	55
FeatureAMP5	Main	16	30	32	22	26	52	5
FeatureAMP6	Playlist	2	12	17	8	7	10	10
	Kernel	8	47	50	29	18	35	14
FeatureAMP8	Application	23	27	41	26	38	72	4
FeatureAMP9	Gui	19	32	53	26	40	92	236
GPL	Graph	11	28	58	5	15	474	23
MinePump	PL_Interface	0	0	0	0	0	0	24
PayCard	PayCard	3	10	5	4	5	7	3
Sudoku	BoardManager	5	16	43	16	14	126	5

TNF: Total number Features that the related class handle; **ScC** Scattering against class features: Counts the number of classes that implement the features that the analyzed class handles; **VarC**: Number of occurrences of variability in the class; **TaC** Tangling in class: Counts the number of context switching features manipulated in the classes; **MFA**: Counts the number of methods of the class that handle the optional features.; **LoCIF**: Counts the number of lines of code for optional features in the analyzed class; **#F**: Number of faults found.

13.92% of the faults are in the class *FeatureAMP* of FEATUTUREAMP3. In summary, we found more than 10 faults in 12 out of 22 faulty classes.

High values for coupling and complexity metrics, such as Response for a Class (RFC) and Weighted Methods per Class (WMC), might be related to fault-prone classes in configurable systems. For all systems with more than one class with faults, WMC and RFC are greater for classes with more faults than classes with fewer faults. For example, in the FEATUREAMP1 system, the *App* class with 406 faults has greater for WMC and RFC than the *PlaylistHandler* class, which has 33 faults. We can also verify how the features are implemented by several classes of the system and mix up with other features through the metrics of features used in this study. We observed that Scattering against class Features (ScC), Number of occurrences of variability in the class (VarC), and Tangling in class (TaC) might be related to fault-prone classes in configurable systems. For example, in FEATUREAMP6, *Kernel* class with 14 faults, has greater SsC, VarC, and TaC values than the *Playlist* class, which 9 faults were identified.

Aiming at investigating whether faulty classes have some characteristics that differ them from non-faulty classes, we compute the Spearman’s rank correlation among all classes in the dataset for all metrics shown in Table 7.1. Due to the small number of failing classes in our dataset, we only found a weak correlation (less than 0.15) between the traditional source code metrics and the faults found. The higher correlations (between 0.10 and 0.15) were found to WMC (0.13), NOF (0.12), NOM (0.11), NOPF (0.11), LoC (0.11), RFC (0.10), and NOPM (0.10). Concerning feature metrics (Table 7.2), we found a slightly highest correlation compared to traditional metrics. However, we were able to find only weak correlations (less than 0.25) between the feature metrics and the faults found. The higher correlation (0.23) was found for TaC. We found a slightly lower correlation to TNF (0.19), LoCIF (0.18), and MFA (0.18) metrics. For the other feature metrics, we found 0.16 for ScC and VarC. These correlations are statistically significant at 99% confidence level (i.e., p-value <0.01). For DIT, NOC, NOSF, and NOSI, no statistically significant correlation was found.

We can see that the feature metrics had a slightly higher correlation with faults than traditional metrics from source code. This result may indicate that classes that are affected by features may be more likely to have feature interaction faults than other classes that are not affected by optional features. Thus, a reduction in the cost of testing would be to test more rigorously only the classes that implement the features with many variability points. In another direction, it is known that classes with high coupling make it challenging to understand the class, which makes the test suite creation more challenging. In this way, the metrics we have pointed out can be good to indicate faults. This result may indicate fault-prone classes of which testers may want to prioritize. In this way, developers can avoid creating classes with high values for these metrics to decrease the chances of feature interaction faults.

Although our fault group is very small, we did not find high correlations for the metrics as indicators of faults. The small percentage of feature interaction faults can be a characteristic of configurable systems. As far as we know, we lack an empirical study to report the average percentage of feature interaction faults in configurable systems at the source code level. The lack of a large dataset with automated tests may be the reason for the lack of studies evaluating the fault location.

Regarding faults at the configuration level, there is a consensus in the literature that faults occur only in a subset of all configurations [Medeiros et al., 2016; Soares et al., 2018a]. Our results indicate that faults also occur in a few classes of configurable systems. Even though in a small amount, these faults are difficult to find and decrease the quality of configurable systems. Our findings might inspire researchers to deeply

investigate characteristics of feature interaction faults considering not only the feature model but also the source code of configurable systems.

RQ₁ Summary. We found that faults are concentrated in only 0.8% of classes of the subject dataset. Two classes are responsible for 49.75% of the faults. Our results also show that source code metrics, such as LoC, WMC, and RFC, and the feature metrics ScC, VarC, and TaC appear to be related to fault-prone classes. Therefore, practitioners should be aware of large, complex, and highly coupled classes and classes that implement features high scattering and tangled.

7.5 Dispersion of Faults over Features (RQ₂)

This section reports the dispersion of faults found over features. Several studies on feature interaction faults analyze the features only concerning the feature model and do not analyze the implementation of features [Hervieu et al., 2011; Johansen et al., 2011; Oster et al., 2011; Al-Hajjaji et al., 2016a]. Hence, there is still a lack of evidence on the characteristics of fault-prone features. To achieve this goal, we compared characteristics of faulty features with characteristics of no faulty features. At the end, we investigated each failed configuration to discover the active features. Looking at the configurable system's source code, we analyze the active features related to the fault location. Hence, we compute the active features in the failed configurations, and we use feature metrics to measure these features. Table 7.3 shows the characteristics of the features frequently active in failed configurations. The main difference of the metrics in this section and metrics from the previous section is that while in this section we measure features concerning the entire configurable system, in the previous section we measure the features concerning each class. For example, in the previous section, we measured scattering and tangling of features per the class. In this section, we use the scattering and tangling to measure each feature in the entire project.

For short, we measure the number of variability points containing other features using the Scattering (Sc.). Some features change the behavior of a more significant amount of code, and others a specific part of the code. In this way, we compute the number of times that features appear in the constructors (Co.) and methods (Me.). The influence of the features in the configurable system through the lines of code (LoC) that the feature handles. Some features cause many points of variability in the source code. We calculate these variability points by feature using the variability points (VP). Some features handle various classes of the configurable system and others only in specific classes. To verify this characteristic, we compute the number of classes that

Table 7.3: Faults found by feature

Name	Feature	Sc.	Co.	Me.	LoC	VP	Ta.	DFT	#F	%F
ATM	DEPOSITING	3	1	17	264	7	3	2	4	5.00
	USER_INTERFACE	3	3	24	237	12	4	3	4	5.00
	WITHDRAWING	1	0	4	89	7	3	3	3	3.75
Bankaccount	OVERDRAFT	1	0	0	6	0	0	2	6	4.17
	BANKACCOUNT	4	2	21	210	2	2	2	6	4.17
	CREDITWORTHINESS	1	0	2	16	2	2	2	3	2.08
	DAILYLIMIT	2	0	4	57	3	2	2	6	4.17
Chess	AI_PLAYER	4	4	38	690	10	2	2	8	100.00
	OFFLINE_PLAYER	3	3	38	663	5	2	2	8	100.00
	ONLINE_PLAYER	3	3	38	665	5	2	2	8	100.00
Companies	LOGGING	4	1	8	187	9	4	2	8	4.17
	PRECEDENCE	4	1	9	186	13	4	2	16	8.33
	TOTAL_WALKER	16	5	32	1119	31	17	3	16	8.33
	TOTAL_REDUCER	16	5	23	1000	31	17	3	1	0.52
	CUT_WHATEVER	29	9	49	1399	31	17	3	17	8.85
	CUT_NO_MANAGER	29	8	46	1345	31	17	3	1	0.52
	GUI	7	5	33	492	48	9	2	9	4.69
FeatureAMP1	PLAYLIST	3	1	13	163	4	2	4	416	45.66
	PROGRESSBAR	1	0	2	24	2	1	5	416	45.66
	SHOWTIME	1	0	2	22	2	1	5	415	45.55
FeatureAMP2	PLAYLIST	3	1	24	321	5	2	3	126	14.03
	PROGRESSBAR	1	0	4	37	4	1	4	129	14.36
	SHOWCOVER	1	0	3	30	3	1	3	90	10.02
FeatureAMP3	REMOVETRACK	1	1	4	54	4	1	5	107	10.09
	VOLUMECONTROL	1	1	4	55	4	1	3	121	11.42
	PLAYLIST	2	2	28	282	8	2	3	182	17.17
	PROGRESSBAR	1	0	6	74	6	1	4	183	17.26
	SHUFFLEREPEAT	2	1	6	141	5	1	5	74	6.98
FeatureAMP4	PLAYER_BAR	9	1	11	154	3	2	2	146	17.16
	PLAYER_CONTROL	2	1	5	47	3	2	3	117	13.75
	PLAYLIST	9	2	29	315	6	4	2	117	13.75
	PROGRESS_BAR	4	1	5	59	3	2	3	125	14.69
FeatureAMP5	PLAYLIST	4	1	25	259	6	1	4	5	0.54
	PROGRESSBAR	1	0	2	23	2	1	5	5	0.54
	SHUFFLEREPEAT	3	0	8	128	3	2	5	3	0.33
FeatureAMP6	REORDER	3	1	7	124	2	2	4	19	1.59
	PROGRESSBAR	2	1	4	86	3	1	4	22	1.84
	PLAYLIST	7	3	31	451	8	5	2	21	1.76
FeatureAMP8	PLAYLIST	1	0	14	274	4	1	3	2	0.23
	QUEUETRACK	1	0	8	292	6	1	4	4	0.45
FeatureAMP9	LOADFOLDER	1	0	5	91	3	1	4	230	26.05
	PLAYLIST	2	1	20	244	6	2	3	230	26.05
	PROGRESSBAR	1	0	3	48	3	1	4	231	26.16
GPL	SEARCH	2	0	6	77	6	2	5	4	1.60
	NUMBER	3	1	4	19	5	3	4	2	0.80
	STRONGLYCONNECTED	4	2	7	66	9	4	4	4	1.60
MinePump	STOPCOMMAND	2	0	2	21	1	1	4	12	18.75
	HIGHWATERSENSOR	2	0	3	27	1	1	4	24	37.50
Paycard	PAYCARD	6	5	16	339	5	4	1	3	50.00
	LOCKOUT	1	0	2	33	2	1	2	3	50.00
	LOGGING	3	2	5	94	3	3	2	2	33.33
Sudoku	COLOR	1	3	0	48	3	1	2	3	15.00
	SOLVER	7	4	19	467	23	7	2	4	20.00
	GENERATOR	5	1	7	195	8	5	2	3	15.00
	STATES	4	1	13	236	13	4	2	4	20.00
	EXTENDEDSDUDOKU	2	0	2	45	2	2	2	2	10.00
	UNDO	2	0	2	44	2	2	2	4	20.00

Sc. (Scattering) column shows the number of classes that the feature manipulates. **Co.**, and **Me.** columns show the number constructors, and methods the feature is inserted into, respectively. **LoC**: Lines of code; shows the number of lines of code handled by the feature. **VP**: variability points, occurrences of variability in source code related to feature. **Ta.** (Tangling) show the number of variability points containing other features. **DFT**: Depth of Feature tree; shows the depth of the feature relative to the feature model. **#F**: The number of faults found where features are present. **%F**: Percentage of the active feature concerning the total configurations analyzed.

the feature manipulates using the Tangling (Ta.). Finally, we use a metric to show the depth of a feature relative to the feature model (DFT). This metric calculates the distance of a feature from the root feature of its feature model.

We analyze the characteristics of the faulty features in the same way used in the faulty classes. We compute the Spearman’s rank correlation among all features in the dataset for all metrics shown in Table 7.3. We investigated a total of 350 features and found values for correlation between 0.41 and 0.66 across all seven metrics in Table 7.3. We have been found for Sc. (0.66), Co. (0.41), Me. (0.64), LoC (0.64), VP (0.64), Ta. (0.65), and DFT (0.66). All these correlations are statistically significant at 99% confidence level (i.e., $p\text{-value} < 0.01$).

The features that stand out with the greatest percentage of faults within the configurations they have high values for the Scattering and Tangling metrics. In this way, the features implemented by several modules of the system and mixed up with other features are good indicators of fault-prone features in configurable systems. We observed that the features with the greatest values for the metrics Sc., Me., LoC, VP, and Ta are more related to faults. These metrics are related to size and measure how much the features influence parts of the configurable system source code. Our results show that the more the feature affected parts of the source code, the more fault-prone this feature will be. This result can improve the set of configurations for testing, as the features that have the most influence on the source code can be prioritized for testing. Thus, researchers might prioritize these features in their testing approaches. On the other hand, practitioners can be more careful on touching features that have greater influence on the configurable system code and they can refactor these features dividing them into multiple features. In this way, configurable system operations will not be concentrated in a small number of features.

RQ₂ Summary. We find that faults are concentrated in small groups of features, and the features that have the most influence on the configurable system are fault-prone. Our results also reveal that feature metrics, such as Scattering and Tangling, seem to be good indicators to identify fault-prone features.

7.6 On the Relation between Faulty Classes and Faulty Features

Figure 7.1 shows the distribution of faults found in the subject dataset. Figure 7.1a shows the occurrence of faults in our dataset and which strategies found faults. Rows indicate systems and columns indicate the testing strategies. We highlight in black the

occurrence of faults for the configurable system. For example, for FEATUREAMP5, only the *YASA-T4* strategy found faults. For FEATUREAMP9, all 16 testing strategies found faults. The only exception, is ATM, of which, only the baselines found faults. In other words, no t-wise strategy found faults on ATM.

Figure 7.1b shows faulty classes and faulty features for all optional code (i.e., code of non-mandatory features). The internal layer shows the configurable systems, the mid layer shows faulty classes, and the external layer shows faulty features. We represent only optional code because they may change across configurations impacting on the testing process. The larger the representation of an instance (i.e., project, class, or feature), the more variation points are related to it. The darker the representation, the larger the number of faults found. For example, in FEATUREAMP4, there are four classes with variability points (*PlayerBar*, *Playlist*, *AudioFactory*, and *FeatureAMP*) and only two of them are faulty. We found 55 and 93 faults in the *PlayerBar* and *FeatureAmp* classes, respectively. The failing code of these classes belongs to four features: `PLAYER_BAR`, `PROGRESS_BAR`, `PLAYER_CONTROL`, and `PLAYLIST`. Faults found in the other classes and features were pointed only by our baselines. Note that a great number of variation points are often related to a great number of faults. For instance, looking at FEATUREAMP3, FEATUREAMP9, FEATUREAMP1, and FEATUREAMP4, we see that they concentrate most variation points and also most faults.

7.7 Threats to Validity

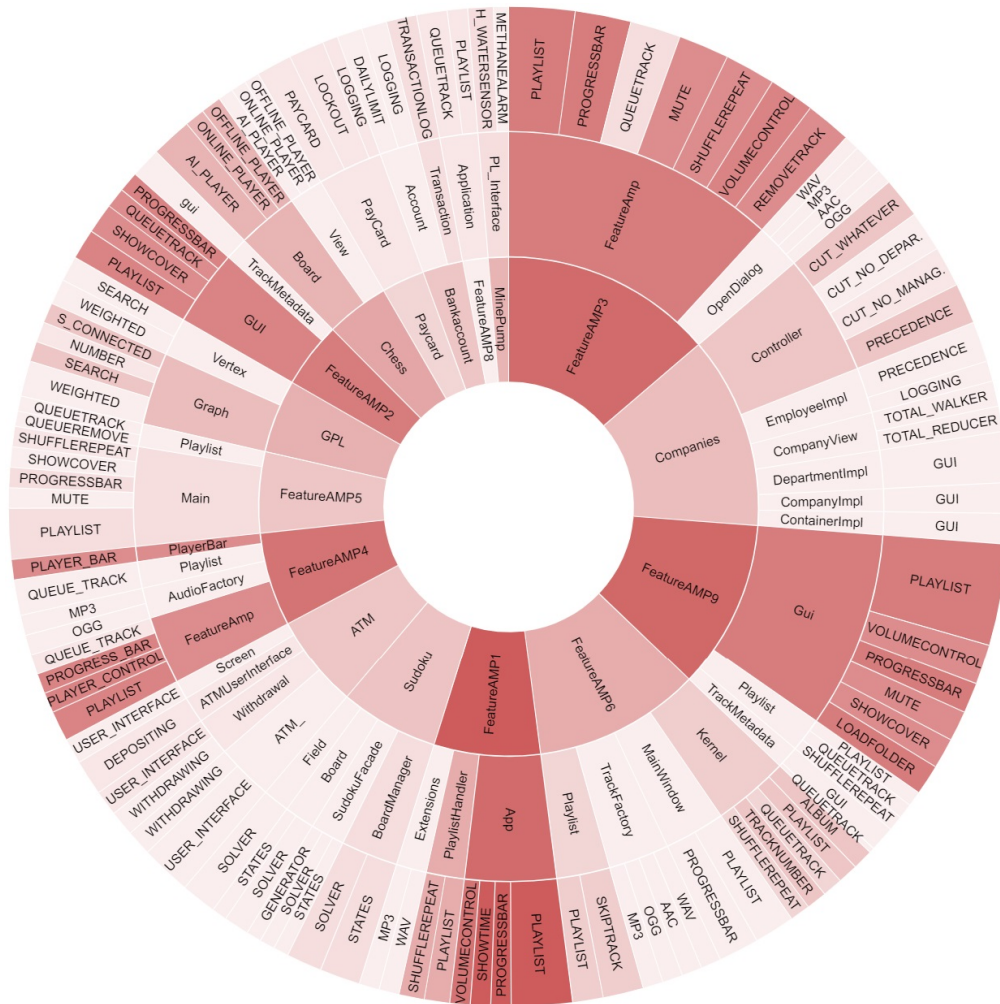
Even with a careful planning, this research can be affected by different factors which might threaten our findings. We discuss these factors and decisions to mitigate their impact on our study divided into external and internal threats to validity below.

External Validity. External validity is threatened mainly by two factors. First, our restriction to variability encoding as variability approach and JAVA as programming language. The generalization to other variability approaches, programming languages, and configurable systems is limited. This limitation of the sample was necessary to reduce the influence of confounds, increasing internal validity [Siegmund and Schumann, 2015]. While more research is needed to generalize to other variability approaches, programming languages, and configurable systems, we are confident that we selected and analyzed a practically relevant variability approach and a substantial number of configurable systems from various domains, longevity, size, and valid configurations. Limiting the programming language is a common limitation of several research studies

7. INVESTIGATING THE DISPERSION OF FAULTS OVER CLASSES AND FEATURES 98

Name	CASA-T1	Chvatal-T1	ICPL-T1	YASA-T1	CASA-T2	Chvatal-T2	ICPL-T2	IncLing-T2	YASA-T2	CASA-T3	Chvatal-T3	ICPL-T3	YASA-T3	CASA-T4	Chvatal-T4	YASA-T4
ATM																
Bankaccount																
Chess																
Companies																
FeatureAMP1																
FeatureAMP2																
FeatureAMP3																
FeatureAMP4																
FeatureAMP5																
FeatureAMP6																
FeatureAMP8																
FeatureAMP9																
GPL																
MinePump																
Paycard																
Sudoku																

(a) Faults distribution found by configurable systems



(b) Faults distribution found by classes and features

Figure 7.1: Faults distribution found

on configurable software systems [Kim et al., 2012b, 2013; Meinicke et al., 2014; Souto et al., 2017; Wong et al., 2018].

Second, our results are restricted to the test suite and faults found in the subject dataset. Using other systems, different versions of the subject systems, or different testing suites may come up with other results. Aiming at minimizing this threat, we chose a dataset previously proposed that follows two reasonable thresholds to increase the quality of the testing suite: 70% of code coverage and 40% of killed mutants. We provided an overview of the dataset in Chapter 4.

Internal Validity. There are two major threats to the internal validity of our study. First, we cannot ensure that we identified all faults present in the subject systems. To increase testing coverage, we used two baselines. While the first baseline runs the test suite in all valid configurations for up to 250 valid configurations sequentially chosen, the second baseline runs the testing suite for up to 250 valid configurations randomly chosen. This way, we run all valid configurations for several subject systems. In addition, our reference list is the union set of all faults found in all configurations run by the sixteen subject testing strategies. Hence, we are not prioritizing faults found by one testing strategy, and all testing strategies have the same chance of finding faults.

Second, we may have not chosen the best metrics to represent systems, features, and classes. We selected several well-known metrics to quantify the size, features, classes, and test suite that compose the configurable systems of the subject dataset. To make this measurement process simpler and automated, we used well-known tools, such as CK TOOL [CK, 2020] and JACOCo [JaCoCo, 2020] and, for the metrics that well-known tools are not able to compute, we create a script. We manually checked the measurement of 8% of the components (e.g., classes and features) to confirm their results. Hence, we believe that similar conclusions would also be achieved using different metrics and tools that quantify similar attributes for the same set of configurable systems and components.

7.8 Final Remarks

In this chapter, we investigated the dispersion of faults over classes and features of 30 configurable software systems. As a result, we found at least one fault in 16 out of 30 systems in the subject dataset. In the analysis, we show that faults are usually concentrated in few classes and features. Moreover, these fault-prone classes and features have distinguish characteristics from fault-free classes and features and these differences can be found based on commonly used source code metrics.

Our results can be used by practitioners to support their decision of know characteristics of classes and features that normally fail in configurable systems, and guide them on increasing test coverage on fault-prone components. Researchers and tool builders may also benefit from our study since we provide several directions for improving existing testing strategies. These directions include, for instance, using source code metrics to identify fault-prone components and use this information on existing testing strategies.

In the next chapter, we report a list of ten challenges faced when performing test suites for configurable systems and dealing with a test suite for our dataset systems.

Chapter 8

Ten Challenges for Testing Configurable Software Systems

Previous chapters presented a sequence of studies about testing configurable software systems. Based on the results of these studies, this chapter proposes a list of ten challenges faced when developing test suites for configurable systems and dealing with a test suite for our dataset systems. Our list of challenges includes, for instance, the challenges of testing high coupled classes and of determining metrics for measuring the quality of the test suite. Our results can be seen as lessons learned on creating tests for configurable systems and they aim at supporting researchers and practitioners on this activity for configurable systems.

Although previous work concentrates on *the explosion of combinations* [Cohen et al., 2003; Kuhn et al., 2004; Kim et al., 2013; Liebig et al., 2013a; Nguyen et al., 2014; Souto et al., 2017]. and *feature interactions* [Kim et al., 2010; Apel et al., 2011; Garvin and Cohen, 2011; Siegmund et al., 2012; Machado et al., 2014; Schuster et al., 2014; Soares et al., 2018b; Nguyen et al., 2019] challenges, we propose other challenges faced when testing configuration systems in practice. For instance, the challenges (1) on the creation of a test suite, (2) on measuring the test suite and its quality, and (3) on the identification of faults are often ignored. We defined ten challenges related to configurable software testing based on our experience when creating and extending the test suite for our dataset systems (Chapter 4). Our main goal is to report the main challenges in the complete life-cycle of 30 open-source configuration systems. Therefore, instead of focus on the well-known challenges present in the literature, we present challenges faced when creating, extending, assessing, and using test suites for 30 configurable systems. For brief, our list of challenges can be summarized as follows.

1. *Creating from scratch and expanding test suites*
2. *Creating test cases in highly coupled classes*
3. *Dealing with the combinatorial explosion of configurations*
4. *Sampling configurations for test*
5. *Running the test suites*
6. *Assessing the quality of the automated test suites*
7. *Measuring the test suites*
8. *Dealing with false positives from tests*
9. *Tracking feature interaction faults to their sources*
10. *Finding technical debts in test cases of configurable systems*

We believe that presenting the faced challenges on the complete life-cycle of 30 configurable systems is just a starting point for researchers to deeply investigate each challenge and propose different ways to deal with them. Practitioners can also benefit from our work by looking at our solutions on each challenge and, hopefully, save time. At the end, we learned that testing configurable systems is related to the ability to automatically run test cases for all possible/desired configurations and identify faults efficiently. Furthermore, a high concentration of faults in a region of code (i.e., a method, class, or feature) might also be an indicative of technical debt in such region (Chapter 6).

The remainder of this chapter is organized as follows. Sections 8.1 to 8.10 present the ten main challenges observed when creating, extending, assessing, using, and identifying faults using the test suites of 30 configurable systems in our previous Chapters 5, 6, and 7. For each challenge, we present our solution. Each following subsection describes a challenge.

8.1 Creating from Scratch and Expanding Test Suites

Both creating and expanding a test suite for configurable systems are challenging because, when designing test cases, the requirements of the configurable system, the variability of features, and their functionality should be taken into account. That is, test suite has to be adjusted in terms of which test cases should be performed according

to the state of the features (active/inactive) since it is not feasible to select each test for the configuration to be tested manually.

The test suite has to be automatically called and only the tests related to the current configuration must be performed. In this way, in the design of the test cases, we have to consider active feature for the test case to be executed. For that, a mechanism must be made in each test case to test the requirements using the specific active features. On the other hand, such a mechanism is necessary to prevent the test cases related to inactive features from being executed. If this occurs, the test result will be a false positive.

Solution. Listing 8.1 shows a test class for the example presented in Listing 1.1 (for ease of viewing, we show Listing 1.1 again in this chapter as Listing 8.2). In addition to the environment set up, it shows a test case testing the `createText` method when the feature `WEATHER` is active. For short, taking “[:weather:]” as input, it should provide “30.0°C” as output. Note that as we have only 2 features, it is easy to create test cases to cover all 4 possible configuration scenarios (i.e., 2^2). However, as the number of related features increases, it is not feasible anymore. To deal with that, we inserted a general configuration file loaded before the execution of the test suite. A further observation is that automated testing tools are not helpful on creating test cases, specially when expanding a test suite. Initially, we tried to use tools to automatically generate test cases (e.g., RANDOOP [Pacheco and Ernst, 2007] and EVOSUITE [Fraser and Arcuri, 2011]). However, after the generation of a small number of test cases, the subsequent new ones usually did not increase the test coverage as well as the percentage of killed mutants. As result, we suggest that testers use these tools carefully to only generate preliminary versions of their test suites.

```

1 public class WeatherReportTest {
2     WeatherReport wr;
3     @Before
4     public void setUp() {
5         wr = new WeatherReport("2019-11-13", "30.0°C");
6     }
7
8     @Test
9     public void weatherTest () {
10        if (Configuration.WEATHER)
11            assertEquals(wr.createText("[:weather:]"), "30.0°C");
12    }
13 }

```

Listing 8.1: Test Cases for the WeatherReport Class

```

1 public class WeatherReport {
2     private Date date;
3     private String temperature;
4
5     public WeatherReport(Date currentDate, String currentTemperature){
6         this.date = currentDate;
7         this.temperature = currentTemperature;
8     }
9
10    public String createText(String c) {
11        if (Configuration.SMILEY)
12            c = c.replace(":", getSmiley(":"));
13        if (Configuration.WEATHER)
14            c = c.replace("[:weather:]", temperature);
15        return c;
16    }
17 }

```

Listing 8.2: Variability encoding example adapted from Meinicke et al. [2016]

8.2 Creating Test Cases in Highly Coupled Classes

To have a deep understanding on the projects of our dataset (Chapter 4), we first needed to read the documentation. After, we spent a considerable time understanding their architecture and source-code. We noted that high coupling classes made the understanding challenging. It also reflected on the creation of test cases because we had to take care of relationships that, in the first moment, were hidden.

Solution. To overcome this scenario, we had to simulate some particular setup that in the first moment was challenging to achieve given high coupling among classes. After creating test cases for several projects, it became easier because we could reuse some strategies, or find some pieces of code that helped with feature interaction problems. We use MOCKITO [Mockito, 2020] to allow the creation of mock object simplifying the development of tests for classes with dependencies of other classes [Spadini et al., 2019].

8.3 Dealing with the Combinatorial Explosion of Configurations

The main challenge researchers and software companies face when developing configurable software systems is to test them. Indeed, adding configuration options yields a combinatorial explosion on the number of configurations to test. Therefore, it is impractical to test them all. For example, a configurable system with 320 features can yield more possibilities (i.e., software variants) than the number of atoms in the universe. It is an insuperable challenge to test all possibilities of configurations for systems with many features. While in traditional software systems there is only one configuration (a combination of features) to be tested, for configurable systems we need to run all tests in several different configurations.

Configurable software testing can be divided into two main parts, generating the configurations and running the automated test suite for these configurations. To generate configurations, we may use strategies that have been developed to test configurable systems (Chapter 5). These strategies can be classified into: variability-aware testing [Kim et al., 2013; Meinicke et al., 2016] and configuration sampling testing [Al-Hajjaji et al., 2016a; Souto et al., 2017]. *Variability-aware testing strategies* explore dynamically all reachable configurations from a given test, by monitoring feature variable accesses during test execution. Concerning running the test suite for the generated configurations, it is possible to make a configuration partition and test the configurations by groups.

Solution. To test the most significant number of configurations in our study, we run all valid configurations for configurable systems with up to 17 features. However, for the other systems, we have created a configuration list for testing that is the union set all configurations run by the sixteen testing strategies. Using multiple sampling strategies allows us to achieve a wide range of configurations at a low cost (Chapter 6).

8.4 Sampling Configurations for Test

Even if testing all possible configurations of a configurable system is possible, the test suite's quality positively impacts the efficiency on observing faults. Generating the configurations and running the test suite is time consuming. Recall that the time consumption can consist of the sum of the time to generate configurations (by sampling strategies) and the time to run the test suite for the configurations (Chapter 6). The ideal scenario is to test all valid configurations but, in practice, the explosion of con-

figurations for testing makes exhaustive testing prohibitively expensive and practically infeasible. Alternatively, developers may test a sample of valid configurations [Al-Hajjaji et al., 2016a]. Several strategies for choosing a sample of configurations to test have been proposed. Some of them use information only from the *feature model* [Al-Hajjaji et al., 2016a; Johansen et al., 2012b], while others also use information from the source code [Kim et al., 2013; Souto et al., 2017].

Solution. Nowadays, several sampling test strategies are available [Al-Hajjaji et al., 2016a; Garvin et al., 2011; Johansen et al., 2012b; Krieter et al., 2020], but, the choice of a testing strategy should be adapted to the time available for testing. Even using a sample strategy testing, the total testing time can take (Chapter 6). In this way, during the configurable system life-cycle, different test strategies can be used to meet the constraints of efficiency and effectiveness. For instance, *1-wise* strategies (e.g., *Chvatal-T1* and *ICPL-T1*) prioritize a minimal number of configurations and are able to find faults. On the other hand, if developers needs a more robust test and can wait longer, they can use strategies that return a greater number of configurations, such as *Chvatal-T4* and *ICPL-T3*. Testing few configurations in the test phase of configurable systems can be an effective alternative as a preliminary and superficial evaluation of the configurable system. However, a preliminary assessment may not discover all feature interaction faults, as these types of faults are harder to discover. In this way, using an efficient strategy can support testing for configurable systems.

8.5 Running the Test Suites

The number of configurations for testing determines the number of times the test suite runs. Therefore, configurable systems with hundreds of configurations result in hundreds of runs of the test suite. Hence, the process of running the suite for each configuration must be automated. Faults in configurable systems can occur in the same test case, but with different configurations. Therefore, there is no point in running the test suite several times if it is impossible to track where the faults occurred.

Solution. We created in our studies a way to record the output of JUnit's execution [JUnit, 2020], always using as a pair the configuration for testing and the line of the test case where the fault was triggered. We use the processor defined in Chapter 5 to record JUnit's execution. In this way, in the subsequent analysis of the faults report, we can quickly identify which configuration fails and which part of the test case code triggered the faults. Algorithm 1 shows a pseudo-code demonstrating how to run the entire test suite for each configuration selected for testing strategy. Algorithm 1

Algorithm 1 Test suite execution manager

Require: List of configurations selected for testing**Ensure:** Test reports

```

1:  $totalConfigurations \leftarrow size(ListOfConfigurations)$ 
2:  $config \leftarrow 1$ 
3: while  $config \leq totalConfigurations$  do
4:    $c = nextConfiguration(ListOfConfigurations)$ 
5:    $setFeatures(c)$ ;
6:   Execute the test suite with configuration  $c$ 
7:    $config \leftarrow config + 1$ 
8: end while

```

has as input a list of configurations for testing. A configurable system test strategy generates this list of valid configurations. Line 5 of Algorithm 1 sets the configurable system features according to the features of each configuration analyzed. Then, in Line 6, we run the test suite for each configuration. Therefore, for all input configurations, Algorithm 1 sets the features according to the analyzed configuration and executes the test suite to test the configurable system according to each configuration. The output of this algorithm is a log generated by Junit composed of the Stack Trace combined with the analyzed configuration.

8.6 Assessing the Quality of the Automated Test Suites

Despite of some techniques proposed for traditional systems to evaluate the quality of tests (e.g., mutation analysis technique), we still face the challenge of testing multiple configurations in configurable systems. *Mutation testing* is a fault-based technique commonly used to evaluate the effectiveness of software testing [Just et al., 2014].

Solution. To use this technique in a test suite for a configurable system, a viable alternative is to choose among the valid configurations, a single configuration that exercises a greater number of test cases. Then, it is possible to use a mutant generation tool, such as PIT [Coles et al., 2016]. Through the generated mutants it is possible to check the quality of the test cases to kill the introduced mutants. New test cases can be created to kill living, non-equivalent mutants [DeMillo et al., 1978]. We use the mutant generation process described in the Chapter 4.

8.7 Measuring the Test Suites

You cannot control something if you cannot measure it DeMarco [1986]. Hence, we needed to find ways to measure what the test suite covers. Metrics like code coverage and the number of test cases are not enough to know if the test suite covers a particular feature. The testers can prioritize the test for specific features because they have more accessible testing features, and testers can do not cover testing other features in the same way. Therefore, we need to measure how the test suite covers the features.

Solution. Specific metrics to identify test suite coverage can also help the tester creating new test cases. For example, if a particular feature has higher priority for users of the configurable system, this feature should be heavily tested. In our studies, we measured the distribution of the test suite for each feature by calculating the variability points for each class and identifying which features reach the variability point. After, we checked which test cases can test each code fragment of the variability point. This way, we were able to estimate how the test suite tested the features.

8.8 Dealing with False Positives from Tests

The identification of false positives requires a manual verification effort [Zolfaghari et al., 2020]. False positives occur mainly due to three problems: (i) poor test case design, (ii) lack of memory, and (iii) timeout. *Poor test case design.* The project of the test suite must consider all the features that are part of the tested functionality. Otherwise, the test case may fail for some configurations on which the target feature is not active. The problem is that this false positive can occur in only a few configurations, giving the impression of the feature interaction faults when a problem occurs in the test case design. Tests can also fail regardless of the configuration. These false positives are easier to identify, but they can cause problems when many faults have to be analyzed. A viable alternative is to correct the defects of these tests first since they fail in all configurations. Thus, with any sample configuration for testing, it is possible to identify these tests.

Lack of memory and Timeout. We analyzed the systems with several sample sizes. We noticed that when running the test suite for many configurations, the execution sometimes crashes due to a lack of memory. To identify problems with lack of memory, we identified the test reports provided by `JUnit`. To deal with this, we ran the test suite for batches of configuration. *Timeout.* Test cases can have problems with timeout definitions that can cause the test case to fail improperly. However, this type of fault is easily identified through the faults report provided by `JUnit`.

Solution. A viable alternative for identifying false positives in tests of configurable systems is to identify the failing configurations reported by the tests. Then, set the features according to the failed configuration, to the configurable system and execute only the failed test case. The analysis must occur initially to verify that all features are in the correct state. Otherwise, we checked if the computational state is correct. If no error is found in the configurable system code, the test case scenario should be reviewed.

8.9 Tracking Feature Interaction Faults to their Sources

Identifying the origin of faults is challenging in configurable systems because the feature interaction must be considered. We perceived that the source code implementing some features are often tangled. We noticed two constraints needed to discover feature interaction faults. First, the sampling of configurations for testing must include at least one configuration with feature interaction faults. Second, at least one test case must identify the feature interaction faults. Failure to comply with these two constraints results in the tests' inability to observe faults. The first constraint takes up the challenge listed in Section 8.3 but, as discussed, it is impracticable to test all possible configurations for systems with several features. Features with feature interaction faults can appear in several valid configurations. However, if only one faulty configuration is found, it is sufficient to solve the problem reflected in several other configurations.

Solution. We observed in our study that a test case can fail several times in the same line of code but with different configurations. The developer solving the feature interaction faults of the configuration found will be correcting the faults in several other configurations that have the same feature combination. In this way, a viable alternative to identify feature interaction faults is to use testing strategies by sampling and creating compelling test suites to find faults.

Furthermore, we suggest to perform, for the list of configurations that caused faults, in each test case, the *frequent itemset mining analysis* [Agrawal et al., 1994]. A frequent itemset is defined as a set of items that occur together in at least a support threshold value of all available transactions. Therefore, we expect to identify which features and their respective states are most present in the faults. It is important to note that this technique can provide itemsets of different sizes, facilitating the analysis.

8.10 Finding Technical Debts in Test Cases of Configurable Systems

Patterns and catalogs have been used in software engineering to deal with challenges. As examples of catalogs, we can cite *design patterns* [Gamma, 1995], *bad smells* [Fowler, 2018], and *architectural patterns* [Buschmann et al., 2007]. We performed this last challenge to encourage the creation of standards and catalogs for testing configurable system. There are dozens of testing strategies that test by sampling [Medeiros et al., 2016; Ferreira et al., 2021, 2019]. However, there is still no consensus on which strategy to use under some constraints, such as time, the criticality of configurable systems, and the number of features.

In addition, a failure taxonomy could be created. In this sense, Soares et al. Soares. [2019] proposed a list of feature interaction faults including Bad Annotation (the expression used in the annotation is incorrect), Wrong Object (an object used in place of another), Misplaced Variable Overwrite (the value of a variable is overwritten in a wrong place with a wrong value), Conditional Statement (incorrect implementation of conditional statements, using wrong or incomplete conditions), and Spread Code (piece of code spread over different features leading to unnecessary dependencies and problems related to code modularity).

Solution. Aiming at comparing sampling testing strategies and understanding the location of faults, we use our dataset and compare suggested configurations from variations of five t-wise testing strategies (Chapters 6). This comparison aims to find which strategies are faster, more comprehensive, effective on identifying faults, time-efficient, and coverage-efficient in this dataset and the reasons why a strategy fared better in one investigated property. As a result, we had a first step toward the identification of patterns and problems in code that may easy or hinder testing.

8.11 Final Remarks

There are dozens of papers that describe the challenge of testing configurable systems [Kim et al., 2013; Medeiros et al., 2016; Meinicke et al., 2016; Souto et al., 2017]. However, previous work has focused on reporting the *combinatorial explosion of the number of configurations to test* and *feature interaction faults*. We summarized the challenges through our observations based on the knowledge acquired from the empirical studies described in the previous chapters.

In this chapter, we presented other challenges faced when testing configuration

systems in practice. For instance, the challenges on the creation of a test suite, on measuring the test suite and its quality, and on the identification of faults are often ignored by the literature. Therefore, we discussed ten challenges related to the life-cycle of tests for configurable systems.

We believe that researchers and practitioners can benefit from the described challenges and can propose solutions for them making our challenges just a starting point for future research. In addition, researchers and practitioners can also benefit from our work by looking at our solutions on each challenge and, hopefully, save time. The next chapter concludes this thesis summarizing the main findings of this thesis and describing paths for future work.

Chapter 9

Conclusion

This chapter summarizes the results of this thesis, regarding its goals, contributions and future work. Section 9.1 summarizes the key findings of this thesis. Section 9.2 reviews our main contributions. Section 9.3 outlines possible ideas for future work.

9.1 Summary

Configurable software systems allow developers to maintain a unique platform and address a diversity of deployment contexts and usages. Testing configurable systems is very challenging due to the number of configurations to run with each test, leading to a combinatorial explosion in the number of configurations and tests. Currently, several testing techniques and tools have been proposed to deal with this challenge, but their potential practical application remains mostly unexplored. The lack of studies to explore the tools that apply those techniques and empirical evaluations including a community-wide dataset motivated us to investigate the literature to find testing tools for configurable software systems and create a dataset of configurable systems with test-enriched.

Considering a gap of knowledge on creating tests for configurable software systems, in this thesis, we conducted a set of empirical studies comparing sampling and sound testing strategies and understanding the location of faults. Then we propose a list of ten challenges faced when performing test suites for configurable systems and dealing with a test suite for our dataset systems. Our results can be seen as lessons learned on creating tests for configurable systems and they aim at supporting researchers and practitioners on this activity for configurable systems. To achieve this goal, the following specific goals (SG) were defined.

- *SG1* Investigate testing tools and strategies for configurable systems in the literature.
- *SG2* Investigate which configurable systems are available in the literature to create a dataset of test-enriched configurable systems.
- *SG3* Perform comparative study with sound testing strategies.
- *SG4* Perform comparative study with t-wise testing strategies.
- *SG5* Analyze at the dispersion of faults found over classes and features in the subject dataset.
- *SG6* Propose a list of ten challenges faced when performing test suites for configurable systems and dealing with a test suite for our dataset systems.

For *SG1*, we conducted a systematic mapping study on testing tools for configurable systems, aiming to verify the state-of-the-art testing tools for configurable systems. Moreover, with this systematic mapping study, we identified the main testing strategies used by these tools (see Chapter 3). Some interesting findings resulted from *SG1* are:

- We found in the systematic mapping study 60 testing tools for configurable systems.
- We analyzed the tools found concerning 16 characteristics and four main testing strategies.
- We presented an overview of 64 primary studies found and presents an overview of how the researchers evaluated testing tools for configurable systems found.
- we discuss our results regarding implications for researchers and practitioners.

For *SG2*, we proposed a test-enriched dataset with 30 configurable software systems. This dataset was created based on a literature review and further implementation of test suites to improve code coverage (see Chapter 4). Some relevant contributions from *SG2* are:

- We searched for configurable systems in the literature and found 243 systems being 60 developed in a Java-based programming language. From those, only 10 systems have a test suite available.

- We created a test suite for other 20 projects. We created tests until they had a coverage of 70% and killed at least 40% of mutants.
- The final dataset has 30 systems varying in domains, size, variability, and test suite size.

For *SG3*, we designed and performed a comparative empirical study of the main sound testing tools (*VarexJ* [Meinicke et al., 2016] and *SPLat* [Kim et al., 2013]) found in the systematic mapping study (see Chapter 3).

- We note that *VarexJ* is generally more efficient than *SPLat*. However, when it was not more efficient, it was by a large difference in specific situations related to its implementation of variability-aware execution.
- We observed that *VarexJ* and *SPLat* presented different results for efficiency while testing the target systems. Although *VarexJ* found more faults than *SPLat* for most of the target systems, such a result deserves a more in-depth investigation because we expected a higher intersection of faults encountered by them.

For *SG4*, we used the community-wide dataset proposed in Chapter 4 and compare recommended configurations from sixteen t-wise testing strategies (eg, *ICPL-T2*, *Chvatal-T4*, and *IncLing -T2*). This comparison aimed to find which strategies are faster, more comprehensive, effective on identifying faults, time-efficient, and coverage-efficient in the community-wide dataset and the reasons why a strategy faced better in one investigated property.

- *1-Wise testing strategies*. Comparing the 1-wise strategies, *ICPL-T1* stood out as the fastest strategy. *Chvatal-T1* was the strategy with the greatest coverage and recall, and was the most time- and coverage-efficient.
- *2-Wise testing strategies*. Comparing the 2-wise strategies, *ICPL-T2* was the fastest and the most time-efficient and got the greatest coverage and recall. *CASA-T2* and *Chvatal-T2* were the most coverage-efficient strategies.
- *3-Wise testing strategies (Figure 6.2c)*. Comparing the 3-wise strategies, *ICPL-T3* was the fastest and the most time-efficient testing strategy. *Chvatal-T3* achieved better coverage, recall, and coverage-efficient.
- *4-Wise testing strategies (Figure 6.2d)*. Comparing the 4-wise strategies, *YASA-T4* was the fastest and the most time-efficient testing strategy. *Chvatal-T4* covered more configurations, achieved the greatest recall and was the most coverage-efficient testing strategy.

For *SG5*, we look at the dispersion of faults found over classes and features in the subject dataset (Chapter 7). Then, we measure each class and feature with metrics commonly used in practice [Chidamber et al., 1998]. Number of lines of code (*LoC*) CK [2020], weighted methods per class (*WMC*) CK [2020], and response for a class (*RFC*) CK [2020] are examples of metrics at class-level. Feature scattering and feature tangling are examples of metrics at feature-level. Finally, we compute Spearman’s rank correlation between the number of faults in a component (i.e., class or feature) and a given metric.

- We found at least one fault in 16 out of 30 systems in the subject dataset.
- We found that faults are concentrated in only 0.8% of classes of the subject dataset and high values of some source code metrics, such as *LoC*, *WMC*, and *RFC*, appear to be related to fault-prone classes.
- We found that faults are concentrated in a few features with high values of some source code metrics, such as feature scattering and tangling.
- We show that faults are usually concentrated in few classes and features. Moreover, these fault-prone classes and features have distinguish characteristics from fault-free classes and features and these differences can be found based on commonly used source code metrics

For *SG6*, we defined ten challenges related to configurable software testing based through our experience when creating and extending the test suite for our dataset systems (Chapter 8). Our main goal is to report the main challenges in the complete life-cycle of 30 open-source configuration systems. Therefore, instead of focus on the well-known challenges present in the literature, we present challenges faced when creating, extending, assessing, and using test suites for 30 configurable systems. For brief, our list of challenges is:

1. *Creating from scratch and expanding test suites*
2. *Creating test cases in highly coupled classes*
3. *Dealing with the combinatorial explosion of configurations*
4. *Sampling configurations for test*
5. *Running the test suites*
6. *Assessing the quality of the automated test suites*

7. *Measuring the test suites*
8. *Dealing with false positives from tests*
9. *Tracking feature interaction faults*
10. *Finding technical debts in test cases of configurable systems*

9.2 Contribution

Although previous work concentrates on *the explosion of combinations* and *feature interactions* challenges. Considering that the lack of studies reporting and investigating the challenges on the complete life-cycle of testing configurable systems, we aim at filling this gap by reporting the challenges we faced when creating, extending, assessing, using, and identifying faults on the test suites of 30 configurable systems. Additionally, we present other challenges faced when testing configuration systems in practice. For instance, the challenges (1) on the creation of a test suite, (2) on measuring the test suite and its quality, and (3) on the identification of faults are often ignored. In addition to the list of ten challenges mentioned, this thesis resulted in the following contributions.

- We documented results of a systematic mapping study of the literature on test tool for configurable systems.
- We proposed a test-enriched dataset with 30 configurable software systems.
- We documented results of an empirical study about two sound test tools.
- We provide evidence of which testing strategies are faster, more comprehensive, more effective on finding faults, more time-efficient, and coverage-efficient based on data from a community-wide dataset.
- We show that faults are usually concentrated in few classes and features. Moreover, these fault-prone classes and features have distinguish characteristics from fault-free classes and features and these differences can be found based on commonly used source code metrics.
- We propose a list of ten challenges faced when performing test suites for configurable systems and dealing with a test suite for our dataset systems. Our list includes, for instance, the challenges of testing high coupled classes and of determining metrics for measuring the quality of the test suite.

9.3 Future Work

We present a list of the main challenges we have identified in conducting the empirical studies described in this thesis. Furthermore, we present at least one solution to the proposed challenges for each challenge and use these solutions throughout the empirical studies described in this thesis. However, our solutions are preliminary and need to be validated in other contexts as another strategy of implementing configurable systems such as annotative. Therefore, it is necessary to expand the scope of our solutions before proper generalization of our results, with additional case studies and experiments comparing these languages other than Java. In the future work in this direction, we plan to investigate different solutions related to tests of configurable systems for other languages and implementation techniques.

We describe our lessons learned from each study described in this thesis. However, an important branch of our work that can be explored in future works is creating specific guidelines for developers, testers, and researchers on testing configurable systems based on lessons learned from this thesis. We believe that the challenges listed in this thesis can be used as input for creating guidelines. For example, the Generating Configurations for Test challenge (see Section 8.4) can be explored to establish guidelines on how to deal with test configurations.

Our restriction to variability encoding as variability approach and JAVA as a programming language is the main limitation of our work. The generalization to other variability approaches, programming languages, and configurable systems is limited. This limitation of the sample was necessary to reduce the influence of confounds, increasing internal validity. While more research is needed to generalize to other variability approaches, programming languages, and configurable systems, we are confident that we selected and analyzed the relevant variability approach and a substantial number of configurable systems from various domains, longevity, size, and valid configurations. Limiting the programming language is a common limitation of several research studies on configurable software systems. Another limitation that is relevant to be further explore in future work is the necessity of a tool to support the test for configurable systems. Researchers and tool builders could benefit from our analysis and lessons learned about testing strategies investigated and the dispersion of faults over classes and features and implement a tool. As an example, we described in Section 7.6, that a tool can consider feature coverage, also takes characteristics of features and classes into account on the selection of configurations for test.

Bibliography

- Aaltonen, K., Ihantola, P., and Seppälä, O. (2010). Mutation analysis vs. code coverage in automated assessment of students' testing skills. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion (OOPSLA)*, pages 153--160.
- Abal, I., Brabrand, C., and Wasowski, A. (2014). 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 421--432.
- Abal, I., Melo, J., Stănciulescu, Ș., Brabrand, C., Ribeiro, M., and Wasowski, A. (2018). Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, pages 10-34.
- Agrawal, R., Srikant, R., et al. (1994). Fast algorithms for mining association rules. In *Proceedings. 20th International Conference Very Large Data Bases, VLDB*, volume 1215, pages 487--499. Citeseer.
- Al-Hajjaji (2016). Al-Hajjaji tool. <https://featureide.github.io/>, Accessed 06-Jul-2021.
- Al-Hajjaji, M., Krieter, S., Thüm, T., Lochau, M., and Saake, G. (2016a). IncLing: Efficient Product-line Testing Using Incremental Pairwise Sampling. In *Proceedings of the 15th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE)*, pages 144--155.
- Al-Hajjaji, M., Meinicke, J., Krieter, S., Schröter, R., Thüm, T., Leich, T., and Saake, G. (2016b). Tool Demo: Testing Configurable Systems with FeatureIDE. In *Proceedings of the 12th International Conference on Generative Programming: Concepts and Experiences (GPCE)*, pages 173--177.

- Al-Hajjaji, M., Thüm, T., Lochau, M., Meinicke, J., and Saake, G. (2016c). Effective product-line testing using similarity-based product prioritization. *Software & Systems Modeling (SoSyM)*, MISSING(MISSING):1–23.
- Al-Hajjaji, M., Thüm, T., Meinicke, J., Lochau, M., and Saake, G. (2014). Similarity-based prioritization in software product-line testing. In *Proceedings of the 18th International Software Product Line Conference-Volume 1*, pages 197–206.
- Ammann, P. and Offutt, J. (2016). *Introduction to software testing*. Cambridge University Press.
- Andrews, J. H., Briand, L. C., and Labiche, Y. (2005). Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering*, pages 402–411. ACM.
- Apel, S., Batory, D. S., Kästner, C., and Saake, G. (2013a). *Feature-Oriented Software Product Lines – Concepts and Implementation*. Springer.
- Apel, S. and Kästner, C. (2009). An overview of feature-oriented software development. *Journal Object Technology*, 8(5):49–84.
- Apel, S., Kolesnikov, S., Siegmund, N., Kästner, C., and Garvin, B. (2013b). Exploring Feature Interactions in the Wild: The New Feature-interaction Challenge. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development (FOSD)*, pages 1–8.
- Apel, S., Leich, T., and Saake, G. (2008). Aspectual Feature Modules. *IEEE Transactions on Software Engineering (TSE)*, 34(2):162–180.
- Apel, S., Rhein, A. v., Wendler, P., Grosslinger, A., and Beyer, D. (2013c). Strategies for Product-line Verification: Case Studies and Experiments. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, pages 482–491.
- Apel, S., Speidel, H., Wendler, P., Rhein, A. V., and Beyer, D. (2011). Detection of Feature Interactions Using Feature-aware Verification. In *Proceedings of the 26th International Conference on Automated Software Engineering (ASE)*, pages 372–375.
- Arcaini, P., Gargantini, A., and Vavassori, P. (2015). Generating tests for detecting faults in feature models. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10.

- Arrieta (2019). Arrieta tool. <https://bit.ly/ArrietaJSS2018&n/a&FeatureIDE>, Accessed 07-Jul-2021.
- Arrieta, A., Wang, S., Sagardui, G., and Etxeberria, L. (2019). Search-Based test case prioritization for simulation-Based testing of cyber-Physical system product lines. *Journal of Systems and Software (JSS)*, pages 1--34.
- Baital (2020). Baital tool. <https://doi.org/10.5281/zenodo.4028454>, Accessed 06-Jul-2021.
- Baranov, E., Legay, A., and Meel, K. S. (2020). Baital: An adaptive weighted sampling approach for improved t-wise coverage. In *In 26th ACM Software Engineering Notes (SIGSOFT)*, ESEC/FSE-11, pages 1--12.
- Basili, V. and Rombach, H. D. (1988). The TAME Project: Towards Improvement-oriented Software Environments. *IEEE Transactions on Software Engineering (TSE)*, pages 758--773.
- Batory, D. (2005). Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the 9th International Conference on Software Product Lines (SPLC)*, pages 7--20.
- Batory, D., Höfner, P., and Kim, J. (2011). Feature interactions, products, and composition. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering (GPCE)*, pages 13--22.
- Bowen, T. F., Dworack, F., Chow, C.-H., Griffeth, N., Herman, G. E., and Lin, Y.-J. (1989). The feature interaction problem in telecommunications systems. In *Seventh International Conference on Software Engineering for Telecommunication Switching Systems, 1989. SETSS 89.*, pages 59--62.
- Bürdek, J., Lochau, M., Bauregger, S., Holzer, A., Von Rhein, A., Apel, S., and Beyer, D. (2015). Facilitating Reuse in Multi-goal Test-Suite Generation for Software Product Lines. In *Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 84--99.
- Buschmann, F., Henney, K., and Schmidt, D. (2007). *Pattern-oriented software architecture, on patterns and pattern languages*, volume 5. John wiley & sons.
- CASA (2014). CASA tool. <https://cse.unl.edu/~citportal/citportal/loadTool?page=casa&id=1>, Accessed 06-Jul-2021.

- Cavarlé, G., Plantec, A., Costiou, S., and Ribaud, V. (2018). A Feature-oriented Model-driven Engineering Approach for the Early Validation of Feature-based Applications. *Science of Computer Programming (SCP)*, 161:18--33.
- Chen, L. and Babar, M. A. (2011). A Systematic Review of Evaluation of Variability Management Approaches in Software Product Lines. *Information and Software Technology (IST)*, 53(4):344--362.
- Chidamber, S. R., Darcy, D. P., and Kemerer, C. F. (1998). Managerial Use of Metrics for Object-oriented Software: An exploratory Analysis. *Transactions on software Engineering (TSE)*, pages 629--639.
- Chvatal, V. (1979). A Greedy Heuristic for the Set-covering Problem. *Mathematics of Operations Research*, pages 233--235.
- CK (2020). CK. <https://github.com/mauricioaniche/ck>, Accessed 07-Set-2020.
- Classen, A., Cordy, M., Schobbens, P., Heymans, P., Legay, A., and Raskin, J. (2013). Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Transactions on Software Engineering (TSE)*, pages 1069--1089.
- CLEMENTS, P. and Northrop, L. (2002). *Software Product Lines: Practices and Patterns*. Addison-Wesley.
- Cohen, M. B., Dwyer, M. B., and Shi, J. (2008). Constructing Interaction Test Suites for Highly-configurable Systems in the Presence of Constraints: A Greedy Approach. *IEEE Transactions on Software Engineering (TSE)*, pages 633--650.
- Cohen, M. B., Gibbons, P. B., Mugridge, W. B., and Colbourn, C. J. (2003). Constructing Test Suites for Interaction Testing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 38--48.
- Coles, H., Laurent, T., Henard, C., Papadakis, M., and Ventresque, A. (2016). Pit: A Practical Mutation Testing Tool for Java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*, pages 449--452.
- ConFTGen (2019). ConFTGen tool. <https://github.com/vhfragal/ConFTGen-tool>, Accessed 07-Jul-2021.
- CoPTA (2019). CoPTA tool. <https://www.es.tu-darmstadt.de/es/team/lars-luthmann/copta-analysis>, Accessed 07-Jul-2021.

- CPA/TIGER (2015). cpatiger tool. <http://forsyte.at/software/cpatiger/>, , Accessed 06-Jul-2021.
- Cruz, D., Figueiredo, E., and Martinez, J. (2019). A Literature Review and Comparison of Three Feature Location Techniques using ArgoUML-SPL. In *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 1--10.
- Czarnecki, K. and Eisenecker, U. (2000). *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley.
- da Mota, P. A., Carmo Machado, I. d., McGregor, J. D., de Almeida, E. S., and de Lemos Meira, S. R. (2011). A Systematic Mapping Study of Software Product Lines Testing. *Information and Software Technology (IST)*, pages 407--423.
- DeMarco, T. (1986). *Controlling software projects: Management, measurement, and estimates*. Prentice Hall PTR.
- DeMillo, R. A., Lipton, R. J., and Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34--41.
- Diniz, J. P., Vale, G., Gaia, F., and Figueiredo, E. (2017). Evaluating Delta-oriented Programming for Evolving Software Product Lines. In *Proceedings of the 2th International Workshop on Variability and Complexity in Software Design (VACE)*, pages 27--33. IEEE.
- Dirk, B. and Keremoglu, M. E. (2011). CPAchecker: A Tool for Configurable Software Verification. In *Proceedings of the 23th International Conference on Computer Aided Verification (CAV)*, pages 184--190.
- Do, H., Elbaum, S., and Rothermel, G. (2005). Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and Its Potential Impact. *Empirical Software Engineering (ESE)*, pages 405--435.
- Engström, E. and Runeson, P. (2011). Software Product Line Testing - A Systematic Mapping Study. *Information and Software Technology (IST)*, pages 2--13.
- FeatureIDE (2014). featureide tool. <https://featureide.github.io/>, Accessed 06-Jul-2021.
- Feng, Y., Liu, X., and Kerridge, J. (2007). A product Line Based Aspect-oriented Generative Unit Testing Approach to Building Quality Components. In *Proceedings*

- of the 31st Annual International Computer Software and Applications Conference (COMPSAC)*, pages 403–408.
- Ferreira, F., Diniz, J. P., Silva, C., and Figueiredo, E. (2019). Testing Tools for Configurable Software Systems: A Review-based Empirical Study. In *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 1--10.
- Ferreira, F., Diniz, J. P., Silva, C., and Figueiredo, E. (2020a). Testing Tools for Configurable Software Systems: A Review-based Empirical Study. http://labsoft.dcc.ufmg.br/doku.php?id=about:testing_tools, Accessed 07-Set-2020.
- Ferreira, F., Vale, G., Diniz, J. P., and Figueiredo, E. (2021). Evaluating T-wise Testing Strategies in a Community-wide Dataset of Configurable Software Systems. *Journal of Systems and Software (JSS)*.
- Ferreira, F., Vale, G., Figueiredo, E., and Diniz, J. P. (2020b). On the Proposal and Evaluation of a Test-enriched Dataset for Configurable Systems. In *Proceedings of the 14th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 1--10.
- Ferreira, F., Vale, G., Figueiredo, E., and Diniz, J. P. (2020c). Test4cs dataset. <https://fischerjf.github.io/TeD4CS/>, Accessed 07-Set-2020.
- Ferreira, F., Vigiato, M., Souza, M., and Figueiredo, E. (2020d). Configurable Software Systems: The Failure Observation Challenge. In *Proceedings of the 24th ACM International Systems and Software Product Line Conference (SPLC)*.
- Ferreira, G., Gaia, F., Figueiredo, E., and Maia, M. (2014). On the use of feature-oriented programming for evolving software product lines—A comparative study. *Science of Computer programming - SCP*, pages 65--85.
- Ferreira, J. M., Vergilio, S. R., and Quinaia, M. (2017). Software Product Line Testing Based on Feature Model Mutation. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 27(05):817–839.
- Ferreira, T. N., Vergilio, S. R., and Kessentini, M. (2020e). Many-objective Search-based Selection of Software Product Line Test Products with Nautilus. In *Proceedings of the 24th ACM International Systems and Software Product Line (SPLC)*, pages 1--4.
- FEST (2020). FEST. <http://code.google.com/p/fest/>, Accessed 07-Set-2020.

- Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Castor Filho, F., et al. (2008). Evolving Software Product Lines with Aspects. In *Proceedings of the ACM/IEEE 30th International Conference on Software Engineering (ICSE)*, pages 261--270. IEEE.
- Fischer, S., Lopez-Herrejon, R., and Egyed, A. (2018). Towards a Fault-detection Benchmark for Evaluating Software Product Line Testing Approaches. In *Proceedings 33th Symposium on Applied Computing (SAC)*, pages 2034--2041.
- Fischer, S., Ramler, R., Linsbauer, L., and Egyed, A. (2019). Automating test reuse for highly configurable software. In *Proceedings of the 23rd International Systems and Software Product Line (SPLC)*, pages 1--11.
- Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.
- Fragal, V. H., Simao, A., Mousavi, M. R., and Turker, U. C. (2019). Extending HSI test generation method for software product lines. *The Computer Journal*, pages 109--129.
- Fraser, G. and Arcuri, A. (2011). Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416--419. ACM.
- Gaia, N., Ferreira, C., Figueiredo, E., and de Almeida Maia, M. (2014). A Quantitative and Qualitative Assessment of Aspectual Feature Modules for Evolving Software Product Lines. *Science of Computer Programming (SCP)*, 96:230--253.
- Galindo, J. A., Turner, H., Benavides, D., and White, J. (2016). Testing Variability-intensive Systems Using Automated Analysis: An Application to Android . *Software Quality Journal (SQJ)*, pages 365--405.
- Gamma, E. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Education India.
- Garvin, B. and Cohen, M. (2011). Feature Interaction Faults Revisited: An Exploratory Study. In *Proceedings of the 22th International Symposium on Software Reliability Engineering (ISSRE)*, pages 90--99.

- Garvin, B. J., Cohen, M. B., Myra, B., and DWYER, M. B. (2011). Evaluating Improvements to a Meta-heuristic Search for Constrained Interaction Testing. *Empirical Software Engineering (ESE)*, 16(1):61–102.
- Gopinath, R., Jensen, C., and Groce, A. (2014). Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*, pages 72--82. ACM.
- Greiler, M., Deursen, A. v., and Storey, M.-A. (2012). Test Confessions: A Study of Testing Practices for Plug-in Systems. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 244--254.
- GrES (2019). GrES tool. <https://drive.google.com/drive/folders/1xumU6qxBesloq69j0PMbpr0aia0KDq82>, Accessed 07-Jul-2021.
- Guidsl (2020). Guidsl Tool. <https://www.cs.utexas.edu/~schwartz/ATS/fopdocs/guidsl.html>, Accessed 07-Set-2020.
- Halin, A., Nuttinck, A., Acher, M., Devroey, X., Perrouin, G., and Baudry, B. (2019). Test Them all, is it Worth it? Assessing Configuration Sampling on the JHipster Web Development Stack. *Empirical Software Engineering (ESE)*, pages 674--717.
- Hasan, I. H., Ahmed, B. S., Potrus, M. Y., and Zamli, K. Z. (2020). Generation and application of constrained interaction test suites using base forbidden tuples with a mixed neighborhood Tabu search. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 30:363--398.
- Henard (2020). Henard tool. http://research.henard.net/SPL/SSBSE_2014/, Accessed 07-set-2020.
- Henard, C., Papadakis, M., Perrouin, G., Klein, J., Heymans, P., and Le Traon, Y. (2014a). Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Transactions on Software Engineering*, pages 650--670.
- Henard, C., Papadakis, M., Perrouin, G., Klein, J., and Traon, Y. (2013a). PLEDGE: A Product Line Editor and Test Generation Tool. In *Proceedings of the 17th International Software Product Line Conference (SPLC)*, pages 126--129.
- Henard, C., Papadakis, M., Perrouin, G., Klein, J., and Traon, Y. L. (2013b). Multi-objective test generation for software product lines. In *Proceedings of the 17th International Software Product Line Conference (SPLC)*, pages 62--71.

- Henard, C., Papadakis, M., and Traon, Y. (2014b). Mutation-based Generation of Software Product Line Test Configurations. In *Proceedings of the 6th International Symposium on Search Based Software Engineering (SSBSE)*, pages 92–106.
- Hervieu, A., Baudry, B., and Gotlieb, A. (2011). Pacogen: Automatic Generation of Pairwise Test Configurations from Feature Models. In *Proceedings of the 22th International Symposium on Software Reliability Engineering (ISSRE)*, pages 120–129.
- Hierons, R. M., Li, M., Liu, X., Parejo, J. A., Segura, S., and Yao, X. (2020). Many-objective test suite generation for software product lines. *Transactions on Software Engineering and Methodology (TOSEM)*, pages 1–46.
- ICPL (2012). ICPL tool. <https://martinfjohansen.com/splc2012/>, Accessed 06-Jul-2021.
- IncLing (2016). IncLing tool. <http://www.tinyurl.com/IncrementalSampling>, Accessed 06-Jul-2021.
- JaCoCo (2020). JaCoCo. <https://www.eclemma.org/jacoco/>, Accessed 07-Set-2020.
- Jia, Y. and Harman, M. (2010). An Analysis and Survey of the Development of Mutation Testing. *IEEE transactions on software engineering (TSE)*, pages 649–678.
- Johansen (2012). Johansen tool. https://research.henard.net/SPL/SPLC_2013/, Accessed 07-Jul-2021.
- Johansen, M. F., Haugen, Ø., and Fleurey, F. (2011). 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)*, pages 638–652.
- Johansen, M. F., Haugen, Ø., and Fleurey, F. (2012a). An Algorithm for Generating T-wise Covering Arrays from Large Feature Models. In *Proceedings of the 16th International Software Product Line Conference (SPLC)*, pages 46–55.
- Johansen, M. F., Haugen, Ø., Fleurey, F., Eldegard, A. G., and Syversen, T. (2012b). 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)*, pages 269–284.

- JUnit (2020). JUnit. <https://junit.org/junit5/>, Accessed 07-Set-2020.
- Just, R., Jalali, D., Inozemtseva, L., Ernst, M. D., Holmes, R., and Fraser, G. (2014). 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*, pages 654–665. ACM.
- Kaltenecker, C., Grebhahn, A., Siegmund, N., Guo, J., and Apel, S. (2019). Distance-based sampling of software configuration spaces. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 1084–1094. IEEE.
- Kastner., C. (2010). *Virtual Separation of Concerns: Toward Preprocessors 2.0*. PhD thesis, Otto-von-Guericke-Universitat Magdeburg.
- Kiczales, G. (1996). Aspect-oriented programming. *ACM Computing Surveys (CSUR)*, (4es):154--es.
- Kim, C. H. P., Batory, D. S., and Khurshid, S. (2011a). Reducing Combinatorics in Testing Product Lines. In *Proceedings of the 10th International Conference on Aspect-oriented Software Development (AOSD)*, pages 57–68.
- Kim, C. H. P., Batory, D. S., and Khurshid, S. (2011b). Reducing Combinatorics in Testing Product Lines. In *Proceedings of the 10th International Conference on Aspect-oriented Software Development (AOSD)*, pages 57–68.
- Kim, C. H. P., Bodden, E., Batory, D., and Khurshid, S. (2010). Reducing Configurations to Monitor in a Software Product Line. In *Proceedings of the 18th International Conference on Runtime Verification*, pages 285–299.
- Kim, C. H. P., Khurshid, S., and Batory, D. (2012a). Shared Execution for Efficiently Testing Product Lines. In *23th International Symposium on Software Reliability Engineering (ISSRE)*, pages 221–230.
- Kim, C. H. P., Marinov, D., Khurshid, S., Batory, D., Souto, S., Barros, P., and d’Amorim, M. (2013). SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 257–267.
- Kim, K., Kim, H., Ahn, M., Seo, M., Chang, Y., and Kang, K. C. (2006). ASADAL: a Tool System for Co-development of Software and Test Environment Based on Product Line Engineering. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 783–786.

- Kim, P., Khurshid, S., and Batory, D. (2012b). Shared Execution for Efficiently Testing Product Lines. In *Proceedings of the 23th International Symposium on Software Reliability Engineering (ISSRE)*, pages 221–230.
- Kitchenham, B. and Charters, S. (2007). Guidelines for Performing Systematic Literature Reviews in Software Engineering. Engineering. In *Technical report, Ver. 2.3 Technical Report (EBSE)*.
- Krieter, S., Thüm, T., Schulze, S., Saake, G., and Leich, T. (2020). YASA: Yet Another Sampling Algorithm. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 1–10.
- Krzysztof, C. and Eisenecker, U. W. (2000). *Generative Programming: Methods, Tools and Applications*. Addison-Wesley.
- Kuhn, D. R., Kacker, R. N., and Lei, Y. (2010). Practical Combinatorial Testing. NIST special Publication Vol. 800.
- Kuhn, D. R. and Reilly, M. J. (2002). An Investigation of the Applicability of Design of Experiments to Software Testing. In *Proceedings of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop.*, pages 91–95.
- Kuhn, D. R., Wallace, D. R., and Jr., A. M. G. (2004). Software Fault Interactions and Implications for Software Testing. *IEEE Transactions on Software Engineering (TSE)*, pages 418–421.
- L., J. F. H., Ferreira, T. N., and Vergilio, S. R. (2018). Incorporating user preferences in a software product line testing hyper-heuristic approach. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8.
- Lamanca, B. P., Polo, M., and Piattini, M. (2015). PROW: A Pairwise Algorithm with ConstRaints, Order and Weight. *Journal of Systems and Software (JSS)*, 99:1–19.
- Lamanca, P., Polo, M., and Piattini, M. (2013). Systematic Review on Software Product Line Testing. In *Software and Data Technologies (ICSOFTE)*, pages 58–71.
- Lee, H., Yang, J.-s., and Kang, K. C. (2012a). VULCAN: Architecture-model-based Workbench for Product Line Engineering. In *Proceedings of the 16th International Software Product Line Conference (SPLC)*, pages 260–264. ACM.
- Lee, J., Kang, S., and Lee, D. (2012b). A Survey on Software Product Line Testing. In *Proceedings of the 16th International Software Product Line Conference (SPLC)*, pages 31–40.

- Lei, Y., Kacker, R., Kuhn, D. R., Okun, V., and Lawrence, J. (2008). IPOG/IPOG-D: Efficient Test Generation for Multi-way Combinatorial Testing. *Software Testing, Verification and Reliability (STVR)*, pages 125--148.
- Li, L., Martinez, J., Ziadi, T., Bissyandé, T. F., Klein, J., and Traon, Y. L. (2016). Mining Families of Android Applications for Extractive SPL Adoption. In *Proceedings of the 20th International Systems and Software Product Line Conference (SPLC)*, pages 271--275.
- Li, X., Wong, W. E., Gao, R., Hu, L., and Hosono, S. (2018). Genetic algorithm-based test generation for software product line with the integration of fault localization techniques. *Empirical Software Engineering*, pages 1--51.
- Liebig, J., von Rhein, A., Kästner, C., Apel, S., Dörre, J., and Lengauer, C. (2013a). Scalable Analysis of Variable Software. In *Proceedings in 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 81--91.
- Liebig, J., Von Rhein, A., Kästner, C., Apel, S., Dörre, J., and Lengauer, C. (2013b). Scalable Analysis of Variable Software. In *Proceedings in 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 81--91.
- Linux (2020). Linux kernel. <http://www.kernel.org>, Accessed 07-Set-2020.
- Lopez-Herrejon, R. E., Ferrer, J., Chicano, F., Haslinger, E. N., Egyed, A., and Alba, E. (2014). Towards a Benchmark and a Comparison Framework for Combinatorial Interaction Testing of Software Product Lines. *The Computing Research Repository (CoRR)*.
- Lopez-Herrejon, R. E., Fischer, S., Ramler, R., and Egyed, A. (2015). A First Systematic Mapping Study on Combinatorial Interaction Testing for Software Product Lines. *Proceedings of the 8th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1--10.
- Lübke, D., Greenyer, J., and Vatlin, D. (2019). Effectiveness of combinatorial test design with executable business processes. In *Empirical Studies on the Development of Executable Business Processes*, pages 199--223.
- Luthmann (2019). Luthmann tool. http://wwwiti.cs.uni-magdeburg.de/iti_db/research/spl-testing/, Accessed 07-Jul-2021.

- Luthmann, L., Gerecht, T., and Lochau, M. (2019a). Sampling strategies for product lines with unbounded parametric real-time constraints. *International Journal on Software Tools for Technology Transfer (STTT)*, pages 613--633.
- Luthmann, L., Gerecht, T., Stephan, A., Bürdek, J., and Lochau, M. (2019b). Minimum/maximum delay testing of product lines with unbounded parametric real-time constraints. *Journal of Systems and Software*, 149:535--553.
- Machado, I., McGregor, J., Cavalcanti, Y., and Almeida, E. (2014). On Strategies for Testing Software Product Lines: A Systematic Literature Review. *Information and Software Technology (IST)*, pages 1183 -- 1199.
- Mao, D., Chen, L., and Zhang, L. (2019). An Extensive Study on Cross-Project Predictive Mutation Testing. In *Proceedings of the 12th Conference on Software Testing, Validation and Verification (ICST)*, pages 160--171.
- Marijan, D., Gotlieb, A., Sen, S., and Hervieu, A. (2013). Practical Pairwise Testing for Software Product Lines. In *Proceedings of the 17th International Software Product Line Conference (SPLC)*, pages 227--235.
- Martinez, J., Assunção, W. K., and Ziadi, T. (2017). ESPLA: A Catalog of Extractive SPL Adoption Case Studies. In *Proceedings of the 21st International Systems and Software Product Line Conference (SPLC)*, pages 38--41.
- Matnei Filho, R. A. and Vergilio, S. R. (2016). A multi-objective test data generation approach for mutation testing of feature models. *Journal of Software Engineering Research and Development*, 4(1):1--29.
- McGregor, J. D. (2001). Testing a software product line. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.
- Medeiros, F., Kästner, C., Ribeiro, M., Gheyi, R., and Apel, S. (2016). A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 643--654.
- Meinicke, J., Thüm, T., Schröter, R., Benduhn, F., and Saake, G. (2014). An Overview on Analysis Tools for Software Product Lines. In *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools-Volume 2 (SPLC)*, pages 94--101.
- Meinicke, J., Wong, C.-P., Kästner, C., Thüm, T., and Saake, G. (2016). On Essential Configuration Complexity: Measuring Interactions in Highly Configurable Systems.

- In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)*, pages 483–494.
- Metrics (2020). Metrics. <http://metrics.sourceforge.net/>, Accessed 07-Set-2020.
- Mockito (2020). Mockito. <https://site.mockito.org/>, Accessed 07-Set-2020.
- Mozilla (2020). Firefox web browser. <http://hg.mozilla.org>, Accessed 07-Set-2020.
- Munoz, D., Oh, J., Pinto, M., Fuentes, L., and Batory, D. (2019). Uniform random sampling product configurations of feature models that have numerical features. In *Proceedings of the 23rd International Systems and Software Product Line (SPLC)*, pages 289–301.
- Nautilus-VTSPL (2018). Nautilus/VTSPL tool. <https://github.com/nautilus-framework/nautilus-plugin-vtspl>, Accessed 06-Jul-2021.
- Nguyen, H. V., Kästner, C., and Nguyen, T. N. (2014). Exploring Variability-aware Execution for Testing Plugin-based Web Applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 907–918.
- Nguyen, H. V. K. C. N. T. N. (2014). Exploring variability-aware execution for testing plugin-based web applications. In *36th Proceedings of the International Conference on Software Engineering (ICSE)*, pages 907–918.
- Nguyen, S., Nguyen, H., Tran, N., Tran, H., and Nguyen, T. (2019). Feature-interaction aware configuration prioritization for configurable code. In *Proceedings of the 34th International Conference on Automated Software Engineering (ASE)*, pages 489–501. IEEE.
- Nie, C. and Leung, H. (2011). A Survey of Combinatorial Testing. *Computing Surveys*, pages 11:1–11:29.
- Offutt, J., Liu, S., Abdurazik, A., and Ammann, P. (2003). Generating test data from state-based specifications. *Software testing, verification and reliability (STVR)*, 13(1):25–53.
- Oster, S., Markert, F., and Ritter, P. (2010). Automated incremental pairwise testing of software product lines. In Bosch, J. and Lee, J., editors, *Software Product Lines: Going Beyond*, pages 196–210, Berlin, Heidelberg. Springer Berlin Heidelberg.

- Oster, S., Zink, M., Lochau, M., and Grechanik, M. (2011). Pairwise Feature-interaction Testing for SPLs: Potentials and Limitations. In *Proceedings of the 15th International Software Product Line Conference, Volume 2 (SPLC)*, page 6.
- Pacheco, C. and Ernst, M. D. (2007). Randoop: feedback-directed random testing for java. In *OOPSLA Companion*, pages 815--816.
- Pacogen (2011). Pacogen tool. <http://www.irisa.fr/lande/gotlieb/resources/Pacogen/Pacogen.html>, Accessed 07-Jul-2021.
- Palix, N., Thomas, G., Saha, S., Calvès, C., Lawall, J., and Muller, G. (2011). Faults in Linux: Ten Years Later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 305--318.
- Papadakis, M., Chekam, T. T., and Le Traon, Y. (2018a). Mutant Quality Indicators. In *Proceedings of the 11th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 32--39.
- Papadakis, M., Shin, D., Yoo, S., and Bae, D. (2018b). Are Mutation Scores Correlated with Real Fault Detection? A Large Scale Empirical Study on the Relationship Between Mutants and Real Faults. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pages 537--548.
- ParTeg (2008). ParTeg tool. <http://parteg.sourceforge.net>, Accessed 06-Jul-2021.
- Pereira, J. A., Constantino, K., and Figueiredo, E. (2015). A Systematic Literature Review of Software Product Line Management Tools. In *Proceedings of the 14th International Conference on Software Reuse (ICSR)*, pages 73--89.
- Petrović, G. and Ivanković, M. (2018). State of Mutation Testing at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE)*, pages 163--171.
- PLEDGE (2014). PLEDGE tool. <https://github.com/christopherhenard/pledge>, Accessed 07-Jul-2021.
- Pohl, K., Böckle, G., and Linden, F. V. D. (2005). *Software product line engineering: foundations, principles, and techniques*. Springer.
- Pohl, K. and Metzger, A. (2006). Software Product Line Testing. *Information and Software Technology (IST)*, pages 78--81.

- Post, H. and Sinz, C. (2008). Configuration Lifting: Verification Meets Software Configuration. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 347--350.
- Ramgouda, P. and Chandraprakash, V. (2019). Constraints handling in combinatorial interaction testing using multi-objective crow search and fruitfly optimization. *Soft Computing*, pages 2713--2726.
- Razali, N. M. and Wah, Y. B. (2011). Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests. *Journal of statistical modeling and analytics*, 2:21--33.
- REFRACT (2014). REFRACT tool. http://cse.unl.edu/~myra/artifacts/Refract_2014/, Accessed 07-Jul-2021.
- Reuling (2019). Reuling tool. <https://pi.informatik.uni-siegen.de/Mitarbeiter/dreuling/sp1c15/>, Accessed 07-Jul-2021.
- Reuling, D., Bürdek, J., Rotärmel, S., Lochau, M., and Kelter, U. (2015). Fault-based product-line testing: Effective sample generation based on feature-diagram mutation. In *Proceedings of the 19th International Conference on Software Product Line (SPLC)*, pages 131--140.
- Riebisch, M. (2003). Towards a more precise definition of feature models. *Modelling variability for object-oriented product lines*, pages 64--76.
- Rocha, L., Machado, I., Almeida, E., Kästner, C., and Nadi, S. (2020). A semi-automated iterative process for detecting feature interactions. In *Proceedings of the 34th Brazilian Symposium on Software Engineering (SBES)*, pages 778--787.
- Ruland (2018). Ruland tool. <https://www.es.tu-darmstadt.de/es/team/sebastian-ruland/gpce18>, Accessed 07-Jul-2021.
- Ruland, S., Luthmann, L., Bürdek, J., Lity, S., Thüm, T., Lochau, M., and Ribeiro, M. (2018). Measuring effectiveness of sample-based product-line testing. In *Proceedings of the 17th International Conference on Generative Programming: Concepts and Experiences (GPCE)*, pages 119--133.
- Samih, H. and Bogusch, R. (2014). MPLM-MaTeLo Product Line Manager:Relating Variability Modelling and Model-based Testing. In *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools-Volume 2 (SPLC)*, pages 138--142.

- Sampling, D.-B. (2019). Distance-Based Sampling tool. https://github.com/se-passau/Distance-Based_Data&XML, Accessed 07-Jul-2021.
- Sánchez, A. B., Segura, S., Parejo, J. A., and Cortés, A. R. (2017). Variability Testing in the Wild: the Drupal Case Study. *Software and Systems Modeling (SoSyM)*, pages 173--194.
- Santos, A., Alves, P., Figueiredo, E., and Ferrari, F. (2016). Avoiding Code Pitfalls in Aspect-oriented Programming. *Science of Computer Programming (SCP)*, 119:31--50.
- Schaefer, I., Bettini, L., Bono, V., Damiani, F., and Tanzarella, N. (2010). Delta-oriented Programming of Software Product Lines. In *Proceedings of the 14 th International Conference on Software Product Lines (SPLC)*, pages 77--91.
- Schaefer, I., Bettini, L., and Damiani, F. (2011). Compositional Type-checking for Delta-oriented Programming. In *Proceedings of International Conference on Aspect-oriented Software development (AOSD)*, pages 43--56.
- Schuster, S., Schulze, S., and Schaefer, I. (2014). Structural Feature Interaction Patterns: Case Studies and Guidelines. In *Proceedings of the 8th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 1--8.
- Seiger, R. and Schlegel, T. (2012). Test Modeling for Context-aware Ubiquitous Applications with Feature Petri Nets. In *Proceedings of the 2th Workshop on Model-based Interactive Ubiquitous Systems (MODIQUITOUS)*.
- Sheskin, D. J. (2020). *Handbook of Parametric and Nonparametric Statistical Procedures*. crc Press.
- Shi, J., Cohen, M. B., and Dwyer, M. B. (2012). Integration Testing of Software Product Lines Using Compositional Symbolic Execution. In *Proceedings International of the 15th Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 270--284.
- Siegmund, J. and Schumann, J. (2015). Confounding Parameters on Program Comprehension: A Literature Survey. *Empirical Software Engineering (ESE)*.
- Siegmund, N., Kolesnikov, S., Kästner, C., Apel, S., Batory, D., Rosenmüller, M., and Saake, G. (2012). Predicting Performance Via Automated Feature-interaction Detection. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 167--177.

- Slaby, J., Strejček, J., and Trtík, M. (2013). ClabureDB: Classified Bug-Reports Database. In *Proceedings of the 14th International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 268–274.
- Smarch+BBPF (2019). Smarch+BBPF tool. https://github.com/se-passau/Distance-Based_Data&XML, Accessed 07-Jul-2021.
- Soares., L. (2019). *Feature Interactions in Highly Configurable Systems: A Dynamic Analysis Approach with VarXplorer*. PhD thesis, Federal University of Bahia (UFBA).
- Soares, L., Meinicke, J., Nadi, S., Kästner, C., and de Almeida, E. (2018a). Varxplorer: Lightweight Process for Dynamic Analysis of Feature Interactions. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 59–66.
- Soares, L. R., Schobbens, P., do Carmo Machado, I., and de Almeida, E. S. (2018b). Feature Interaction in Software Product Line Engineering: A Systematic Mapping Study. *Information and Software Technology (IST)*, 98:44–58.
- Souto, S. (2015). *Addressing high dimensionality and lack of feature models in testing of software product lines*. PhD thesis, Federal University of Pernambuco (UFPE).
- Souto, S., d’Amorim, M., and Gheyi, R. (2017). Balancing Soundness and Efficiency for Practical Testing of Configurable Systems . In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, pages 632–642.
- Spadini, D., Aniche, M., Bruntink, M., and Bacchelli, A. (2019). Mock objects for testing java systems. *Empirical Software Engineering*, (3):1461–1498.
- SPL2go (2020). SPL2go. <http://spl2go.cs.ovgu.de/projects/>, Accessed 07-Set-2020.
- SPLAT (2013). SPLAT tool. <https://github.com/sabrinadfs/splat-sampling>, Accessed 07-Jul-2021.
- SPLtool (2013). SPLtool tool. <http://martinfjohansen.com/models2011/spltool/>, Accessed 07-Jul-2021.
- SPLVerifier (2011). SPLVerifier tool. <http://fosd.net/FAV>, Accessed 07-Jul-2021.
- Svahnberg, M., Van Gorp, J., and Bosch, J. (2005). A Taxonomy of Variability Realization Techniques. *Software: Practice and experience (SPE)*, (8):705–754.

- Swanson, J., Cohen, M. B., Dwyer, M. B., Garvin, B. J., and Firestone, J. (2014). Beyond the Rainbow: Self-adaptive Failure Avoidance in Configurable Systems. In *Proceedings of the 22th International Symposium on Foundations of Software Engineering (SIGSOFT)*, pages 377–388.
- Sánchez (2012). Sánchez tool. <https://www.isa.us.es/~isaweb/anabsanchez/material.zip>, Accessed 07-Jul-2021.
- Sánchez, A. B., Segura, S., and Ruiz-Cortés, A. (2014). A Comparison of Test Case Prioritization Criteria for Software Product Lines. In *Proceedings of the 7th International Conference on Software Testing, Verification and Validation (ICST)*, pages 41--50.
- TEMSA (2015). TEMSA tool. <http://zen-tools.com/TEMSA/>, Accessed 07-Jul-2021.
- TESALIA (2016). TESALIA tool. <http://tesalia.github.io>, Accessed 07-Jul-2021.
- Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., and Leich, T. (2014). FeatureIDE: An Extensible Framework for Feature-oriented Software Development. *Science of Computer Programming (SCP)*, pages 70--85.
- Thüm, T., Schaefer, I., Apel, S., and Hentschel, M. (2012). Family-based Deductive Verification of Software Product Lines. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 11--20.
- Uzuncaova, E., Garcia, D., Khurshid, S., and Batory, D. (2008). Testing Software Product Lines Using Incremental Test Generation. In *Proceedings of the 19th International Symposium on Software Reliability Engineering (ISSRE)*, pages 249–258.
- Vale, G. and Figueiredo, E., Abílio, R., and Costa, H. (2014). Bad Smells in Software Product Lines: A Systematic Review. In *Proceedings of the 8th Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*, pages 84--94. IEEE.
- Vale, G., Abílio, R., Freire, A., and Costa, H. (2015a). Criteria and Guidelines to Improve Software Maintainability in Software Product Lines. In *Proceedings of the 12th International Conference on Information Technology-New Generations (ITNG)*, pages 427--432. IEEE.
- Vale, G., Albuquerque, D., Figueiredo, E., and Garcia, A. (2015b). Defining Metric Thresholds for Software Product Lines: A Comparative Study. In *Proceedings of the 19th International Conference on Software Product Line (SPLC)*, pages 176--185.

- VarexC (2018). VarexC tool. <https://github.com/chupanw/vbc>, Accessed 06-Jul-2021.
- VarexJ (2016). VarexJ tool. <http://meinicke.github.io/VarexJ/>, Accessed 07-Jul-2021.
- Varshosaz, M., Al-Hajjaji, M., Thüm, T., Runge, T., Mousavi, M. R., and Schaefer, I. (2018). A Classification of Product Sampling for Software Product Lines. In *Proceedings of the 22nd International Systems and Software Product Line*, pages 1--13.
- VarXplorer (2018). VarXplorer tool. <https://github.com/larirsoares/VarXplorer>, Accessed 06-Jul-2021.
- Wang, S., Ali, S., and Gotlieb, A. (2013). Minimizing Test Suites in Software Product Lines Using Weight-based Genetic Algorithms. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1493–1500.
- Wang, S., Ali, S., and Gotlieb, A. (2015). Cost-effective Test Suite Minimization in Product Lines Using Search Techniques. *Journal of Systems and Software (JSS)*, 103(3):370–391.
- Weißleder, S., Sokenou, D., and Schlingloff, B. (2008). Reusing state machines for automatic test generation in product lines. In *1st workshop on model-based testing in practice (MoTiP)*, page 19.
- Wohlin, C. (2014). Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 1--10.
- Wohlin, C., Mendes, E., Felizardo, K. R., and Kalinowski, M. (2020). Guidelines for the search strategy to update systematic literature reviews in software engineering. *Information and Software Technology (IST)*.
- Wong, C., Meinicke, J., Lazarek, L., and Kästner, C. (2018). Faster Variational Execution with Transparent Bytecode Transformation. *Proceedings of the ACM on Programming Languages (OOPSLA)*.
- Xiang, Y., Huang, H., Li, M., Li, S., and Yang, X. (2021). Looking For Novelty in Search-based Software Product Line Testing. *IEEE Transactions on Software Engineering (TSE)*, pages 1--1.

- Yan, L., Hu, W., and Han, L. (2019). Optimize SPL test cases with adaptive simulated annealing genetic algorithm. In *Proceedings of the ACM Turing Celebration Conference-China*, pages 1--7.
- YASA (2020). YASA tool. https://github.com/FeatureIDE/FeatureIDE/tree/config_generation, Accessed 06-Jul-2021.
- Yu, L., Lei, Y., Nourozborazjany, M., Kacker, R. N., and Kuhn, D. R. (2013). An Efficient Algorithm for Constraint Handling in Combinatorial Test Generation. In *Proceedings of the 6th International Conference on Software Testing, Verification and Validation (ICST)*, pages 242--251.
- Zolfaghari, B., Parizi, R. M., Srivastava, G., and Haleimariam, Y. (2020). Root causing, detecting, and fixing flaky tests: State of the art and future roadmap. *Software: Practice and Experience*, pages 1--17.