Universidade Federal da Bahia
Instituto de Matemática e Estatística

Programa de Pós-Graduação em Ciência da Computação

FEATURE INTERACTIONS IN HIGHLY CONFIGURABLE SYSTEMS: A DYNAMIC ANALYSIS APPROACH WITH VARXPLORER

Larissa Rocha Soares

TESE DE DOUTORADO

Salvador, Bahia – Brasil
21 de Fevereiro de 2019

LARISSA ROCHA SOARES

FEATURE INTERACTIONS IN HIGHLY CONFIGURABLE SYSTEMS: A DYNAMIC ANALYSIS APPROACH WITH VARXPLORER

Esta Tese de Doutorado foi apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Orientador: Prof. Dr. Eduardo Santana de Almeida
Co-orientador: Prof. Dr. Christian Kästner e Prof. Dr. Ivan do Carmo Machado

Salvador, Bahia – Brasil
21 de Fevereiro de 2019

Ficha catalográfica elaborada pelo Sistema Universitário de Bibliotecas (SIBI/UFBA),
com os dados fornecidos pelo(a) autor(a).

Rocha Soares, Larissa
   Feature Interactions in Highly Configurable
Systems: a dynamic analysis approach with VarXplorer
/ Larissa Rocha Soares. -- Salvador, 2019.
   195 f. : il

   Orientador: Eduardo Santana de Almeida.
   Coorientador: Ivan do Carmo Machado.
   Tese (Doutorado - Programa de Pós Graduação em
Ciência da Computação) -- Universidade Federal da
Bahia, Instituto de Matemática e Estatística, 2019.

   1. Software Engineering. 2. Feature Interaction.
3. Configurable Software. 4. Variability-Aware
Execution. 5. Software Product Lines. I. Santana de
Almeida, Eduardo. II. do Carmo Machado, Ivan. III.
Título.

LARISSA ROCHA SOARES

"FEATURE INTERACTIONS IN HIGHLY CONFIGURABLE SYSTEMS: A DYNAMIC ANALYSIS APPROACH WITH VARXPLORER"

Esta tese foi julgada adequada à obtenção do título de Doutor em Ciência da Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação da UFBA.

Salvador, 21 de fevereiro de 2019.

[illegible]

Prof. Dr. Eduardo Santana de Almeida
Orientador/PGCOMP

[illegible]

Prof. Dr. Ivan do Carmo Machado
Co-Orientador/PGCOMP

Christina von Flach G. Chavez

Prof.ª Dr.ª Christina von Flach G. Chavez
Membro interno/PGCOMP

Rodrigo Rocha Gomes e Souza

Prof. Dr. Rodrigo Rocha Gomes e Souza
Membro interno/PGCOMP

[illegible]

Prof. Dr. Márcio de Medeiros Ribeiro
Membro externo/Universidade Federal de Alagoas

Eduardo Figueiredo

Prof. Dr. Eduardo Magno Lages Figueiredo
Membro externo/Universidade Federal de Minas Gerais

Gostaria de dedicar esta tese ao senhor Jorge José Araujo Rocha, meu tio e padrinho que há 2 anos nos deixou de forma inesperada e foi morar ao lado do Pai. Obrigada por tanto amor e carinho. Um dia estaremos juntos de novo.

ACKNOWLEDGEMENTS

During the PhD., I had the amazing opportunity of being received by professor Christian Kästner at Carnegie Mellon University (CMU), Pittsburgh, US. I do not have enough words to thank Christian for all the support and fantastic talks during the year I spent at Pittsburgh. CMU was a watershed in my life. I'm not the same person as before. I learned so much. I've matured a lot, both professionally and as a person. The way I use to research has changed. My way of thinking has also changed and I am very grateful for all of it.

I also would like to thank Christian for introducing me Sarah Nadi, who have been working with us since I got at the US. Thank you, Sarah, for your generosity, kindness and partnership. Still, I would like to thank Christian's research group, specially the one who became my friend, Jens Meinicke, for the infinite talks, support during hard times, laugh and friendship. Last, and not least, I would like to thank all my friends in Pittsburgh. Without them, my life would have been much more lonely and sad. They were my family for 1 year and have brought companionship and joy into my life. Most of the happiest moments I had, I had because of you. Thank you, girls.

Agora em português. Querido Deus, obrigada por tudo que o Senhor tem feito por mim e pelos meus. O Teu amor cobre as minhas fraquezas e a Tua fidelidade é maior do que todos os obstáculos na minha vida. Mais uma vez, obrigada.

A vida de ninguém é fácil e muito menos parecida com o que é postado nas redes socias. Nem tudo é festa. Coincidência ou não, os anos do doutorado foram os mais difíceis da minha vida, I mean, vida pessoal. Mas, tudo passa. Eu precisava seguir em frente, e valeu muito a pena. Vieram momentos inesquecívies, muito felizes, journals, papers, conferências internacionais, pessoas especiais e lugares maravilhosos. Porém, sozinha eu não conseguiria. Nunca.

Eu gostaria de agradecer ao meu orientador no Brasil, o professor Eduardo Almeida. Obrigada por todo suporte durante esses anos, por acreditar e confiar em mim. Gostaria de agradecer também ao professor Ivan Machado. Muito obrigada por todas as nossas conversas, por ser sempre tão gentil e pela disponibilidade. Sempre após as nossas reuniões, eu me sentia muito melhor. Obrigada de verdade. Eu também preciso falar do melhor grupo de pesquisa e laboratório do mundo, RiSE Labs e Lab INES! A companhia diária, o ambiente descontraído, as risadas e todos os momentos com vocês serviram como combustível para que eu pudesse seguir em frente. Magno, Gau, Leo, Iuri, Michele, Jonatas, Paulo, Renata, Tassio, Alberto, Crescêncio. Obrigada.

Um muito obrigada também aos meus pais, que mesmo sem entender direito o que eu fazia, estavam sempre lá me dando todo o suporte necesssário e todo o amor desse mundo. Tudo que fiz e faço é por vocês e pra vocês, sempre. Preciso também agradecer

ao meu noivo e melhor amigo, Igor, e agradecer aos meus sogros por estarem sempre ao meu lado. Obrigada! Em especial, obrigada, Pig, não só por todo amor e carinho, mas também por toda ajuda no doutorado em si. Bug de programação, a quem eu recorreria? E quem leria meus artigos? E pra cada artigo, ele lia todas as versões várias e várias vezes. Discutia comigo, me dava ideias e estava sempre lá, à disposição. Não tenho palavras para te agradecer. Eu te amo muito.

Finalmente, preciso agradecer ao meu irmão, sobrinhos (Davi e Bia), meus primos, tios, tias e amigos. Obrigada por acreditarem em mim quando nem eu mesma acreditava. Amo muito vocês.

Livros não mudam o mundo, quem muda o mundo são as pessoas. Os livros so mudam as pessoas.

—MARIO QUINTANA

RESUMO

Sistemas altamente configuráveis (também conhecidos como linhas de produtos de software) fornecem oportunidades significativas de reuso, uma vez que eles adaptam variantes do sistema com base em um conjunto de *features*. Essas features podem interagir de formas indesejadas, resultando em falhas. Além disso, a maioria das interações não é facilmente detectável, já que especificações de interações entre features geralmente não são definidas, especificadas e documentadas em um projeto de software.O problema da interação entre features tem sido um assunto desafiador por anos. Apesar da existência de estudos que mapeiam essas interações, ainda não há muitos trabalhos sobre a compreensão de estratégias, atividades, artefatos e lacunas de pesquisa para interações em sistemas configuráveis. Desta forma, esta tese provê inicialmente um mapeamento sistemático de estudos por meio da análise de 40 trabalhos, os quais foram classificados de acordo com os estágios do ciclo de vida de desenvolvimento e a solução de interação apresentada (detecção ou resolução de interações).Análises recentes têm focado na detecção de erros de interação de features a partir de especificações globais, ou seja, especificações que todas as configurações de um sistema configurável precisam cumprir. No entanto, especificações no nível de features ou interações são geralmente negligenciadas e raramente documentadas. Neste cenário, muitas abordagens não conseguem detectar todos os problemas de comportamento do sistema, especialmente erros não cobertos por especificações globais e erros que não resultam em uma falha ou outro comportamento facilmente observável.Ao invés de partir de um conjunto de especificações como a maioria das abordagens, propomos inspecionar as interações de features à medida que são detectadas e classificá-las gradativamente como benignas ou problemáticas. Nossa abordagem e ferramenta, VarXplorer, fornece um processo de inspeção que ajuda os desenvolvedores a distinguir as interações intencionais das interações que podem levar a bugs. Usamos a execução variacional para observar interações internas ao fluxo de controle e fluxo de dados de sistemas altamente configuráveis e propomos gráficos de interação de features como uma representação concisa de todas as interações entre pares de features.Por fim, realizamos dois estudos empíricos para avaliar como o processo de inspeção e os gráficos de interação de features podem ajudar os desenvolvedores a identificar e entender interações suspeitas. O primeiro é um experimento controlado que investiga e compara a capacidade dos desenvolvedores ao identificar interações suspeitas com e sem o VarXplorer. O segundo foca no processo iterativo de execução de casos de teste e como ele proporciona uma análise de interações mais rápida e objetiva.

Palavras-chave: Interação entre features; Software configurável; Especificação de interações entre features; Execução ciente de variabilidade;

ABSTRACT

Highly configurable systems (as known as software product lines) provide significant reuse opportunities by tailoring system variants based on a set of features. Those features can interact in undesired ways which may result in faults. However, most interactions are not easily detectable as specifications of feature interactions are usually missing. The feature interaction problem has been a challenging subject for years. Despite the existence of studies to map out available evidence on feature interaction for single systems development, there is a lack of understanding on common strategies, activities, artifacts and research gaps for interactions in configurable systems. Thus, this thesis initially gathered systematic mapping study evidence by analyzing 40 feature interaction primary studies, which were classified according to development lifecycle stages and the feature interaction solution presented, either detection, resolution or general analysis. Recent analyses focused on detecting feature interaction bugs from global specifications, i.e., specifications that all configurations of a configurable system need to fulfill, such as requiring that each configuration does not crash. However, specifications at the feature level are usually missing and, then, many approaches may not detect all incorrect system behavior, specially bugs not covered by global specifications and bugs that do not result in a crash or other easily observable behavior. Instead of starting from a set of specifications like most approaches, we propose to inspect feature interactions as they are detected and incrementally classify them as benign or problematic. We aim to provide an inspection process that helps developers to distinguish intended interactions from interactions that may lead to bugs. We use variational execution to observe internal interactions on control and data flow of highly configurable systems. To help developers understand these interactions, we propose feature-interaction graphs as a concise representation of all pairwise interactions. We provide two analyses that provide additional details about interactions, namely suppress and require interactions. Our approach and tool, VarXplorer, provide an iterative analysis of feature interactions allowing developers to focus on suspicious cases. Finally, we perform two empirical studies to evaluate the inspection process and how feature interaction graphs can help users identify suspicious interactions. The first study is a controlled experiment to investigate and compare the ability of users when identifying suspicious interactions with and without VarXplorer, in a setting composed of different systems, performing different tasks. The second study focuses on the iterative process of test cases execution and how it can be used for a faster and more objective feature interaction analysis.

Keywords: Feature Interaction; Configurable Software; Feature Interaction Specification; Variability-Aware Execution;

CONTENTS

V Conclusions

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

SPL	Software Product Lines
FOSD	Feature-Oriented Software Development
FDA	Domain analysis
FDD	Domain design and specification
FDI	Domain implementation
FPC	Product configuration and generation
De	Early Detection
Dsc	Source Code Detection
Re	Early Resolution
Rsc	Source Code Resolution
Ae	Early Analysis
Asc	Source Code Analysis
ICFI	Feature Interactions in Telecommunications and Software Systems
ICSE	International Conference on Software Engineering
SPLC	International Systems and Software Product Line Conference
ASE	International Conference on Automated Software Engineering
ESEC/FSE	European Software Engineering Conference and Symposium on the Foundations of Software Engineering
VaMoS	International Workshop on Variability Modelling of Software-Intensive Systems
GPCE	International Conference on Generative Programming: Concepts Experiences
OOPSLA	International Conference on Object- Oriented Programming, Systems, Languages, and Applications

FOSD Workshop	International Workshop on Feature-Oriented Software Development
BDD	Binary Decision Diagram

PART I

OVERVIEW

Chapter 1

INTRODUCTION

Highly-configurable systems, such as software product lines, provide significant reuse opportunities by tailoring system variants based on a set of *features* (aka. configuration options) [31]. Such systems may be composed of thousands of features. For example, the Eclipse IDE[1] has more than 1,600 plugins [31] and the Linux kernel[2] has more than 15,000 configuration options [32, 33]. This large set of options may be combined in different ways, and developers must guarantee that all valid combinations work properly. A common problem in highly configurable systems is that a *feature interaction* between two or more features may result in a surprising behavior that is not easily deduced from the analysis of each feature separately [34]. Even if a system behaves as expected most of the time, it may exhibit unexpected and unwanted interactions under specific feature combinations. The chapter consists of eight sections:

Section 1.1 introduces and motivates this study;

Section 1.2 presents and discusses the objectives of the thesis;

Section 1.3 describes our research questions;

Section 1.4 presents the research design, which involves background, dynamic approach, and empirical studies;

Section 1.5 presents the main contributions of this work;

Section 1.6 defines the topics out of the scope; and

Section 1.7 finally presents the organization of the thesis.

[1]¡https://www.eclipse.org/¿
[2]¡https://www.kernel.org/¿

1.1 MOTIVATION

A software product can be seen as a configuration of features that need to be composed together without violating their particular requirements. On the one hand, it is relatively simple to specify the behavior of a feature in isolation. On the other hand, specifying and detecting interactions among features may not be a straightforward task. Henceforth, we use the term feature to refer to any configuration option, module, or component in a configurable system.

Determining the influence of feature interactions on the system's behavior has been a challenging subject for decades [35]. Anticipating and specifying all likely consequences of each possible feature interaction might not be possible, mainly due to the facts that (i) the number of configurations and feature interactions grows exponentially in relation to the number of features [36]; (ii) the behavior of some interactions may be unknown and unpredictable in advance [34]; and (iii) human effort is required, but people usually do not like writing specifications.

To address those challenges, recent analyses focus on detecting feature interaction bugs from *global specifications*, i.e., specifications that all configurations of a configurable system needs to fulfill, such as requiring that each configuration does not crash [37]. Usually, these approaches check global specifications based on systematic sampling [38, 39, 40], combinatorial interaction testing [41, 42, 43], model checking [44, 45, 46, 47, 48], or variational execution [49, 50, 51, 52].

Additionally, the problem is that features may interact in many ways, for example, by triggering events that enable other features, having control over the same variables, and enforcing conditions that suppress other features [53]. However, since specifications at the feature level are usually missing, the mentioned approaches may not detect all incorrect system behavior, especially bugs not covered by global specifications and bugs that do not result in a crash or other easily observable behavior.

Thus, it is hard to reason about interactions without feature specifications. When statically detected in the source code, (i) predicted interaction may never appear during system execution; (ii) many feature interactions can be observed only at runtime; and (iii) it is difficult to automatically determine if an unexpected interaction is either benign or represent a real problem. Although current dynamic approaches [49, 50, 51, 52] overcome drawbacks of static analysis by analyzing systems at runtime, identifying problematic interactions still remains challenging, especially when no feature specification is provided.

1.2 OBJECTIVES

In this work, we have two main objectives, as follows:

1.2.1 Presenting the state-of-the-art on feature interactions

We investigate highly configurable systems to leverage how do the existing approaches deal with feature interactions in the community. Particularly, we aim to identify common practices and research trends. To accomplish such a goal, we carried out a systematic mapping study to investigate state-of-the-art approaches and identify research topics that researchers and practitioners could address. The systematic mapping study included a set of seven research questions, in which the 35 studies found are mainly classified regarding the feature interaction solution presented, i.e., either detection, resolution, or general analysis of interactions.

1.2.2 Supporting developers on the detection of suspicious feature interactions with VarXplorer

Features are frequently combined to cooperate to an intended behavior (expected interactions). However, most interactions cannot be predicted upfront. Thus, instead of upfront specifications as most of the approaches do, we propose to inspect feature interactions as they are detected and incrementally classify them as either benign or problematic. We aim to provide an inspection process that helps developers to distinguish intended interactions from unintended interactions. This is an automated, tool supported, process. VarXplorer is an approach automated by an Eclipse plug-in. It was developed to analyze control and data flow interactions, besides presenting features relationships, such as the suppression of one feature by another.

1.3 RESEARCH QUESTIONS

On the basis of such defined goals, we established the following research questions that drive this investigation:

- **What is the state-of-the-art on feature interaction research for product lines?**
  During the last decade, research strategies on Software Product Lines (SPL) have been published in the literature regarding detection [12, 13], resolution [54], and analysis [14, 55] of feature interaction. Despite the existence of studies to map out available evidence on feature interaction for single systems development [56], it is still missing a study to SPL. We have conducted a systematic mapping study to investigate strategies, activities, artifacts and research gaps for interactions in software product lines.

- **How can we detect unexpected feature interactions and classify them as either benign or problematic in a highly configurable system?**
  There exists a lot of work aiming to detect faults caused by feature interactions, as well as techniques to resolve them. However, detecting unexpected feature interactions that do not lead to a crash (at least not for the given test cases), but that cause faulty behavior, remains an open challenge. In our work, we aim to address the challenge of helping developers to dynamically identify potentially faulty feature interactions without counting on upfront specifications. We propose an approach that provide an iterative and incremental analysis of interactions and a tool to detect the interactions and relationships between features.

- **How to evaluate such approach?**
  To achieve a comprehensive research rigor and relevance, our tool should be empir-

ically evaluated and results have to be reported to stakeholders, detailing benefits and drawbacks of the approach. Thus, we conducted two complementary studies. First, we performed a controlled experiment to measure the effort to identify a buggy interaction compared to the state-of-the-art tool. We also performed a qualitative analysis based on video and audio recordings, and post-treatment interviews. In the second study, we explored the iterative approach and specifications. We analyzed whether running sequential test cases and using feature-interaction specifications can make it easier to identify suspicious interactions.

1.4 RESEARCH DESIGN

This section describes the research design employed in this work. We split this investigation in three main parts: *Background*; *Dynamic approach*; and *Empirical studies*. Figure 1.1 shows a diagram with these macro parts and an overview of the sub-activities, which we detail next.

Background. The first part presents an overview of the basic concepts that guide this thesis, such as SPL, feature, feature-oriented software development, and feature interactions. In addition, it also encompasses the literature review on feature interaction in product lines.

We first define SPL, main concepts, and objectives. Then, we present key aspects of the feature-oriented software development, which has been widely used in the SPL engineering. Finally, we discuss feature interaction in SPL engineering. We bring the different ways that an interaction is defined, besides presenting the feature interaction classification. Such concepts provide the ground for us to devise our research questions and to narrow down the possibilities to be included in this investigation.

In addition, we perform a systematic mapping study to serve as an in-depth analysis of the current existing knowledge on the comprehension of feature interactions. The mapping classifies the studies according to their proposed solutions, feature interactions types, software lifecycle, software domains, and empirical assessment methods. We also present the studies in 3 different categories: detection, resolution, and general analysis of feature interactions.

Dynamic approach on feature interactions. The second part comprises the proposal of the dynamic approach to detect interactions. As a means of better understanding our proposed approach (VarXplorer), we first discuss the strategies to detect interactions, focusing on the dynamic analysis. To make it easier to explain suspicious interactions and why we should detect them, we also present a running example. Then, we show the preliminary steps of our approach, such as the test case execution strategy and the variability-aware interpreter used to run the systems.

Next, we present details of VarXplorer, which is a dynamic iterative and interactive approach to detect suspicious interactions. It consists of an approach and an Eclipse plug-in that implements the proposed approach. VarXplorer provides information on how features impact the control and data flow of the program. Our tool supports developers with a feature-interaction graph that visualizes this information, mainly showing suppress and require relations between features. Thus, it is shown how we detect interactions and

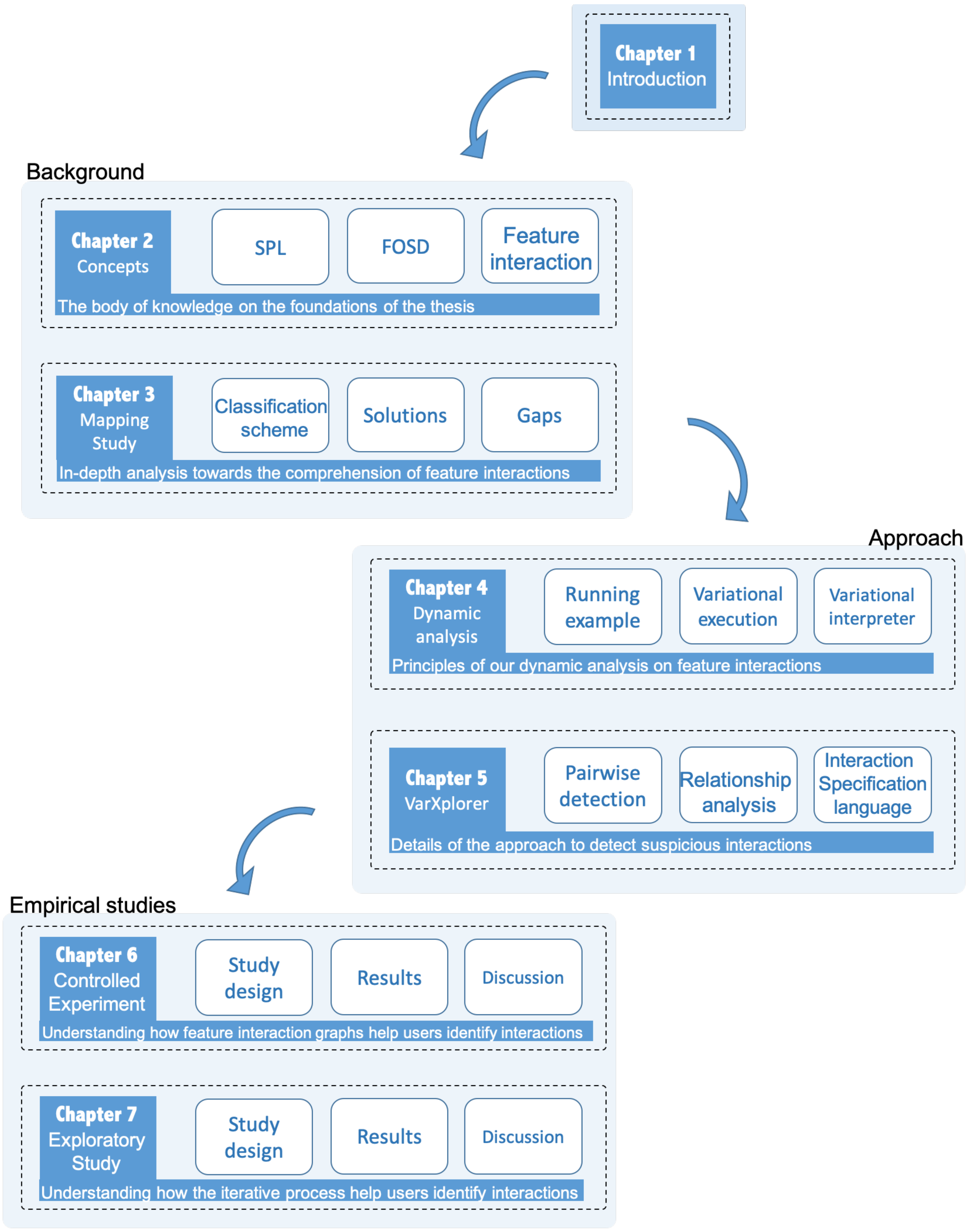

Chapter 1
Introduction
Background
Chapter 2
Concepts
SPL
FOSD
Feature interaction
The body of knowledge on the foundations of the thesis
Chapter 3
Mapping Study
Classification scheme
Solutions
Gaps
In-depth analysis towards the comprehension of feature interactions
Approach
Chapter 4
Dynamic analysis
Running example
Variational execution
Variational interpreter
Principles of our dynamic analysis on feature interactions
Chapter 5
VarXplorer
Pairwise detection
Relationship analysis
Interaction Specification language
Details of the approach to detect suspicious interactions
Empirical studies
Chapter 6
Controlled Experiment
Study design
Results
Discussion
Understanding how feature interaction graphs help users identify interactions
Chapter 7
Exploratory Study
Study design
Results
Discussion
Understanding how the iterative process help users identify interactions

Figure 1.1: Schematic overview of the thesis structure.

relationships between features.

Empirical studies. The third part encompasses two empirical evaluations: (i) a study on feature-interaction graphs to determine if they do help developers identify problematic interactions; and (ii) a study to understanding how the iterative process and feature-interaction specifications may support developers during the testing process.

We conduct a controlled experiment with 24 participants from both academia and industry backgrounds, and measure the effort to identify a buggy interaction based on the information provided by the feature-interaction graph. We also perform an in-depth qualitative analysis based on video and audio recordings, and post-treatment interviews. Then, we carry out a second study to analyze whether the iterative process on individual tests proposed by VarXplorer is able to reduce the complexity of identifying interactions.

1.5 CONTRIBUTIONS

In accordance with our goals, the main contributions of this work are related to our feature interaction approach and they are listed in the following:

1. a *systematic mapping study* to investigate the state-of-the-art in feature interactions. The mapping study is published in the Information and Software Technology journal;
2. a set of *gaps on feature interaction in SPL* area, which we leverage and discussed in the mapping study;
3. a way to *dynamically* detect interactions based on both control and data flow;
4. two classes of interactions (relationships between features), namely *suppress* and *require* interactions. Those classes provide details on how features interact to support developers in *identifying unintended interactions*;
5. *feature-interaction graphs*, a concise visual representation of feature interactions identified at runtime using variational execution;
6. a *feature interaction specification* language to allow and forbid interactions on data and control flow;
7. an *iterative and interactive approach* to refine feature-interaction graphs using feature interaction specifications;
8. an *Eclipse plug-in* to generate feature interaction specifications and remove interactions that do not represent a bug, allowing the developer to focus only on suspicious cases;
9. a *controlled experiment* showing that feature-interaction graphs improve the efficiency of understanding feature interactions compared to the state-of-the-art;
10. an in-depth qualitative analysis showing advantages of the graph components towards the detection of suspicious interactions;

11. an *exploratory study* presenting how the VarXplorer iterative process improves and facilitates the identification of suspicious interactions.

Table 1.1 shows a list of the publications related to the thesis topic in order to get an overview of our contributions so far.

Table 1.1: Publications during the Ph.D. research.

Paper Title	**Venue**	**Year**
Thesis related publications		
1. Feature interaction in software product line engineering: A systematic mapping study [57]	**Infor. and Software Technology Journal**	*2018*
2. VarXplorer: Lightweight Process for Dynamic Analysis of Feature Interactions [58]	**VaMoS**	*2018*
3. VarXplorer: reasoning about feature interactions [59]	**ICSE Student Competition**	*2018*
4. Exploring Feature Interactions without Specifications: A Controlled Experiment [60]	**GPCE**	*2018*
5. Feature interactions in highly configurable systems, an approach and two studies	**TOSEM** [***under work***]	-
Other publications		
6. Non-Functional Properties in Software Product Lines: A Reuse Approach [61]	**VaMoS**	*2015*
7. Analysis of Non-functional Properties in Software Product Lines: A Systematic Review [62]	**EUROMICRO/SEAA**	*2014*
8. SPLICE: A Lightweight Software Product Line Development Process for Small and Medium Size Projects [63]	**SBCARS**	*2014*

1.6 OUT OF SCOPE

It is rather important to define the scope of this thesis. Given all the described before, we consider as out of the scope the following topics:

- *variability-aware interpreter:* we extended an existing interpreter that is well-recognized by the community to execute the system test cases. Since we rely on tooling previously developed, which is open source and published in different venues [50, 64, 65], we assume it is reasonably well developed.

- *test case suite:* there is a number of studies discussing on how to get the best coverage for a test case suite. We do not address this issue in our investigation in order to reduce the scope. We assume that we already have the test cases for the systems used on the running example and evaluation systems;

- *static analysis of feature interactions:* dynamic analysis has many advantages compared to static analysis, such as: detection of real interactions, reduction of false positives, and identification of bugs hardly detected on source code. Although we understand that static approaches have their benefits as well, we chose to focus on the dynamic strategy.

1.7 ORGANIZATION OF THE THESIS

This thesis is structured in five parts and two appendices. Figure 1.1 shows a schematic overview of the thesis structure. Apart from the Introduction Part, the remainder can be outlined in the following way:

Part II - Background. This part provides background concepts on the topics involved in this investigation, as discussed in the Section 1.4. In addition to the basic concepts, it also presents a mapping study on feature interactions for SPL.

Chapter 2 (Concepts) Basic concepts regarding the topic of this thesis.

Chapter 3 (Systematic Mapping Study) We present a literature review on feature interactions.

Part III - Dynamic approach on feature interactions. This part motivates and define in detail the novel strategy to handle feature interaction in SPL engineering. We elaborate on how it was conceived and present how we support developers on the detection of suspicious interactions.

Chapter 4 (On the detection of feature interactions) Definition and discussion of the main concepts used in our approach.

Chapter 5 (VarXplorer) Description of the details on how VarXplorer detects interactions, besides presenting the interactive and iterative approach.

Part IV - Empirical studies. This part presents two empirical studies on the gathering evidence regarding the usefulness of VarXplorer, interaction graphs, iterative process, and feature-interaction specification, as a strategy to identify feature interaction bugs in programs.

Chapter 6 (Controlled Experiment) Planning of the experiment and results.

Chapter 7 (Exploratory Study) Planning of the exploratory study and results.

Part V - Conclusions. Finally, this part concludes the thesis document.

Chapter 8 (Conclusions) Thesis summary, concluding remarks and future work.

PART II

BACKGROUND

Chapter 2

MAIN CONCEPTS AND FOUNDATIONS

SPL engineering defines a set of systems that share common features and artifacts to achieve high productivity, quality, market agility, low time to market, and cost [66]. An SPL product is derived from a configuration of features which need to be compounded together without violating their particular specifications. While it is easy to identify the behavior of a feature in isolation, specifying and resolving interactions among features may not be a straightforward task. The feature interaction problem has been a challenging subject for decades.

The goal of this chapter is to present the basic concepts in the context of this thesis. The chapter consists of four main sections:

Section 2.1 introduces *Software Product Lines*;

Section 2.2 presents the basics of *Feature-Oriented Software Development*, as a mainstream strategy to deliver Software Product Lines;

Section 2.3 presents the definition of *feature interactions* in SPL engineering and how it is usually classified;

Section 2.4 concludes this chapter.

Since the topics of the sections are broad, along this chapter we provide background information rather than introducing all the existing literature.

2.1 SOFTWARE PRODUCT LINES (SPL)

A *feature* describes a unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option [67]. Products of highly configurable systems (respectively, SPL) can be composed by selecting a set of features. Then, SPL can be described as a family of systems created and developed from features. SPL engineering explores the commonalities and manages variabilities among related products, in which it is possible to establish a common platform on top of software assets that can be systematically reused and assembled into different products.

Based on the selection of features, software engineers can configure distinct products satisfying a range of common and variable features, which comprise both functional and non-functional properties [68]. In SPL engineering, features can be organized in a feature model, a graphical representation including variability relations, features constraints and dependencies. Features in a feature model are usually classified as [69]: (i) mandatory, a feature that must be selected whenever its parent feature is selected; (ii) optional feature, a feature that may or may not be selected; (iii) *OR* feature group, when one or more features in the group must be selected; and (iv) *XOR* (alternative) feature group, when one and only one of the features in the group must be selected.

There are several paradigms to develop an SPL, such as: Feature-Oriented Software Development (FOSD), aspect-oriented software development, and component based software engineering [70]. Among these, FOSD is an emerging paradigm that enables customization, and synthesis of software products [70]. The FOSD paradigm has been used in the SPL development in order to take advantage of the systematic application of features in all phases of software lifecycle.

2.2 FEATURE-ORIENTED SOFTWARE DEVELOPMENT (FOSD)

The software development process based on FOSD relies on the concept of a feature to analyze, design, as well as to implement software systems. The FOSD paradigm corresponds to a collection of methods, tools, languages and formalisms connected by features. Basically, FOSD involves four phases: (i) Domain analysis (FDA), (ii) Domain design and specification (FDD), (iii) Domain implementation (FDI), and (iv) Product configuration and generation (FPC) [70]. Since the feature concept is spread through all those phases, approaches for reducing feature interactions problems are also discussed over them.

Feature modeling is the main activity of the FDA phase, whose objective is to identify variabilities and commonalities in a given domain. The architecture of an SPL is designed in the FDD phase, through either formal or informal specifications and modeling languages. In FDI, features are developed to meet its specifications. Finally, the FPC phase is responsible for generating a software product according to user's requirements. FOSD encourages an automatic software product generation based on tools that support a valid features selection. There are several commercial and academic tools available to assist software engineers in finding a valid selection and support a product development, such as: Gears [71], pure::variants [72], FeatureIDE [73], and EASy-Producer [74].

2.3 FEATURE INTERACTION

Feature interaction has been widely discussed in the telecommunications domain. During the 80's [56], many solutions covering different lifecycle stages emerged in that community. A widespread feature interaction example is related to the features *call waiting* and *call forwarding* of a telephone system. When both are present at the same time, the system behavior is ill-defined. *Call waiting* allows managing two interleaved calls – one is suspended while another one is being answered. *Call forwarding when busy* requires to specify a phone number to forward new calls that arrive when the phone is busy. If these features are used together, and a new call is received when the phone is busy, it does not know how to proceed: it can either forward or suspend the new call. Feature interaction can cause unexpected situations, and may even go against the system specification [75].

Although the feature interaction problem came to light in the telecommunications domain, it has been recognized as a general problem in different fields, as follows: in software engineering, to predict, detect and resolve interactions [76]; cyber-physical systems, to reduce CO_2 emissions [77]; in automotive systems domain, contradictory physical forces that lead to unsafe behaviour [78]; and comprehension of feature interaction for the Internet of Things [79].

In our work, we focus on software engineering, which proved to be effective for software production in large scale [80]. While an SPL may comprise a huge number of features, the number of possible interactions is even exponentially bigger. A single feature in an SPL product may interact with another feature or a set of features, creating a high-degree complex interaction. Moreover, a feature behavior may be, for example, dependent on presence conditions (e.g., the busy condition on the telephone example aforementioned) and on input configurations (e.g., user entries, unpredictable variables' values), which can further increase the complexity of detecting interactions. Thus, even if a product behaves as expected most of the time, in specific conditions it might present unexpected interactions on either data or control flow.

Solutions to the feature interaction problem can find a very wide range of applications, including in the domains of smart home systems [15], automotive systems [81], email systems [14], embedded medical devices [7], antivirus products [82], databases [12, 25], web systems [55], network [14], among others.

2.3.1 Feature interaction in SPL engineering

In SPL engineering, feature interaction is usually defined by means of a feature *behavior*, i.e., changes in the behavior of the features involved in an interaction, which do not occur when the features are used in isolation. Feature interaction can also manifest in *non-functional* attributes, for instance when features in combination have an influence on a particular attribute, such as performance [25]. Also, feature interaction can be seen through the *user's* point of view. For example, for a system to be perfectly configured and ready to be delivered, the software engineer must guarantee that all the feature interactions respect the user's intentions.

The literature presents many definitions for feature interaction in the SPL engineering context. Those definitions usually approach three major aspects: feature behaviour, non-functional attributes, and user point of view. The main definitions are described next:

- Abal et al. [83]: "Features in a configurable system interact in non-trivial ways, in order to influence each others functionality. When such interactions are **unintended**, they induce bugs that manifest themselves in certain configurations but not in others".

- Apel et al. [84]: “A feature interaction occurs when the **behavior** of one feature is influenced by the presence of another feature (or a set of other features)”.
- Apel and Kästner [70]: “A feature interaction is a situation in which two or more features exhibit **unexpected behavior** that does not occur when the features are used in isolation”.
- Atlee et al. [13]: “feature interaction is a discrepancy between a feature’s **behavior** in isolation versus its behavior in the presence of other features.”
- Hall [85]: “the desired behavior of a feature combination may violate the ideal of individual feature modularity. However, combining independently designed and validated features often leads to **undesirable behaviors** as well. When this occurs, it is termed a feature interaction.”
- Mosser et al. [54]: “feature interaction as the identification of a mismatch between the **intention of the user** and the obtained product.”
- Schobbens et al. [86]: “A recurrent problem is the one of feature interaction: adding new features may modify the operation of already implemented ones. When this modification is **undesirable**, it is called a **feature interference**.”
- Schuster et al. [12]: “Feature interactions describe the common observation that two or more features (functionally) interact so that the **behavior** of the underlying program may be changed”.
- Siegmund et al. [25]: “An interaction occurs when a particular feature combination has an **unexpected** influence on **performance**.”

In summary, a feature interaction can be described as a situation in which a feature influences another feature in either a positive or negative way. Without an earlier analysis, interactions would only be discovered during feature composition, and even with early detection, it still needs resolution, which usually requires domain knowledge.

Although most definitions present interactions as unintended and undesirable properties, not every interaction is harmful to the system. Sometimes, features are combined to cooperate and accomplish their tasks. Priorities and overridings are some of the strategies used by developers to specify the intended resolution of a feature interaction [81]. However, a solution for a feature interaction issue may require an special attention, i.e., it should neither violate the specifications of the features involved in the interaction nor the specifications of the other system’s features. Configuring products with distinct features that present a correct interaction and offer stable and functional services is essential to ensure the development of a reliable SPL.

2.3.2 Classification

Feature interaction has been categorized under different aspects, such as, according to the software development stage in telecom [87]; based on how they can be detected [85]; and regarding the order and visibility of an interaction [84].

Ohta and Harada [87] represented telecommunication services specification as finite state machines (FSM) and described the interaction problem from the standpoint of the FSM. Hall [85] classified interactions in three categories: (type I) the features dictate contradictory behaviours; (type II) there is no immediate contradiction, but an intended property of one participating feature will eventually be violated, and (type III) other unwanted interaction. Hall provided approaches and tools to detect categories I and II, and gave a support to understand category III.

Plath and Ryan [88] refine Hall's type II, according to the feature to which the violated property belongs to. They also introduce lack of commutativity between features as interaction type IV. Later, Apel et al. [84] classified feature interactions by means of two dimensions: order and visibility. Whereas the order reflects the number of the features involved in the interaction, the visibility describes feature interactions as either *external* (if they impact the user-visible behaviour of the system) or *internal* (if it breaks internal properties of the system or requires specific interaction code). In this study, we follow Apel et al. [84] classification.

External feature interactions can be classified in two categories: functional and non-functional interactions [84]. The former corresponds to interactions that violate the functional specification of a composed system; and the latter refers to interactions that influence non-functional properties of a composed system, such as performance, reliability and security.

Internal feature interactions can be classified into *structural* or *operational* interactions [84]. Features interact *structurally* when *coordination code* is necessary to deal with the problem caused by the interaction. A coordination code represents an additional piece of code responsible for resolving the interaction, which is supplementary to the code combination of the individual features involved. Coordination code can be surrounded by preprocessor directives involving several features, or written in lifters [89] or derivatives [90]. Features interact *operationally* when the data flows or control flows differ from the combination of the flows of the features involved.

2.4 CHAPTER SUMMARY

In this chapter, we presented an overview of this thesis' topic. We started by introducing features in Software Product Lines. We also presented FOSD as a common strategy to develop products in an SPL. Then, we introduced feature interactions and presented how the SPL enginering community define and classify them.

Next chapter presents a mapping study that surveyed existing research on feature interaction in SPL engineering in order to identify common practices and research trends.

Chapter

3

SYSTEMATIC MAPPING STUDY

The goal of this chapter is to present a systematic mapping study of the thesis topic. A mapping study is a way to investigate the state-of-the-art and identify research topics that researchers and practitioners could address. Recently, a variety of systematic mapping studies have been conducted to different SPL fields in order to investigate, for example, SPL testing [91], adoption [92], agile methods [93], non-functional properties [62], and traceability [94].

This systematic mapping study aims at synthesizing existing research related to feature interaction solutions for SPL engineering. Hence, we categorize interactions according to seven research questions. The study focuses on the large amount of research that has been developed in the last years. We analyzed studies published between the years 2004 and 2018, although no lower limit has been set. As a result, 40 studies were found to be relevant and mainly classified regarding to the SPL development lifecycle stages and the feature interaction solution presented, either detection, resolution or general analysis.

Thus, the chapter consists of five main sections:

Section 3.1 presents the systematic mapping study protocol in detail;

Section 3.2 describes the classification scheme adopted in this study and results;

Section 3.3 discusses the mapping study findings and implications of research;

Section 3.4 presents the main threats to the validity of our study; and

Section 3.5 describes gaps and directions for future research.

3.1 MAPPING STUDY PROCESS

A systematic mapping study aims at presenting an overview of a research area, providing its amount of studies, publication frequency over the years, results and trends [1]. It is also used to provide a visual sampling and a classification of studies, besides identifying

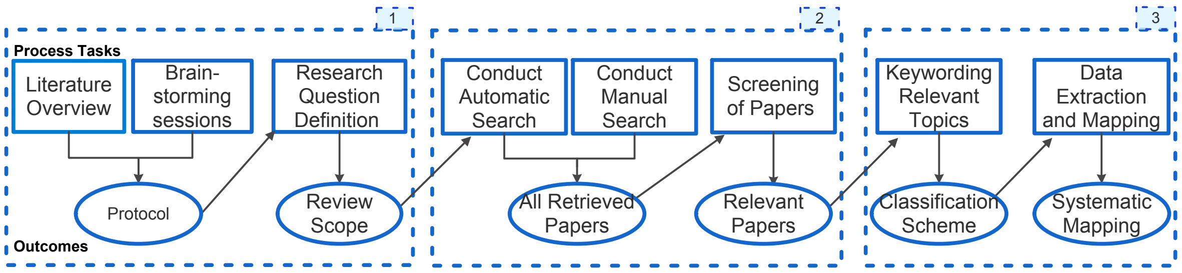


Figure 3.1: The systematic mapping process, adapted from Petersen et al. [1]

publishing forums and research gaps. According to Kitchenham et al. [95], the mapping study's goal is to survey the available knowledge about a topic.

A systematic mapping study process comprises three main phases: planning, conducting and documenting [95]. The planning phase encompasses the development of a protocol, i.e., a framework that includes all the tasks of the mapping study and serves as a guide in the other two phases. In order to design our mapping study protocol, we conducted meetings and brain-storming sessions with SPL researchers, and all this information supported the process and protocol specifications.

Figure 3.1 shows the tasks performed in the mapping study process and the outcomes of each phase. In the first, the protocol and research questions are defined; the second corresponds to the execution of the mapping and it is responsible for searching and screening the papers. Two strategies were used to search for papers: automatic search and manual search, and they were based on a set of inclusion and exclusion criteria. In the final phase, we defined a classification scheme and extracted data mainly based on the research questions. The results regarding a detailed analysis of each primary study is presented as a systematic mapping study. In the next sections, all the process tasks are detailed.

3.1.1 Research Questions

As previously stated, the objective of our mapping study is to identify common practices on feature interactions, research trends, open issues and topics for future research. Thus, we focused on identifying **how do the existing approaches deal with feature interactions in SPL**, which is the general question that drives the research. Hence, five more specific questions were derived, and a summary is shown in Figure 3.2.

- *RQ 1. Which feature interaction solutions have been proposed for SPL engineering?* The solutions presented by the research papers represent the different approaches to handle feature interaction in an SPL development project. They involve detection, resolution and management of feature interactions. In this question, we aim to investigate these existing solutions, which can indicate the aspects addressed by both research and industry communities, besides potential open rooms for improvement.

- *RQ2. What are the different types of feature interactions the approaches deal with?* To answer this question, we used the Apel et al. [84] visibility classification. This

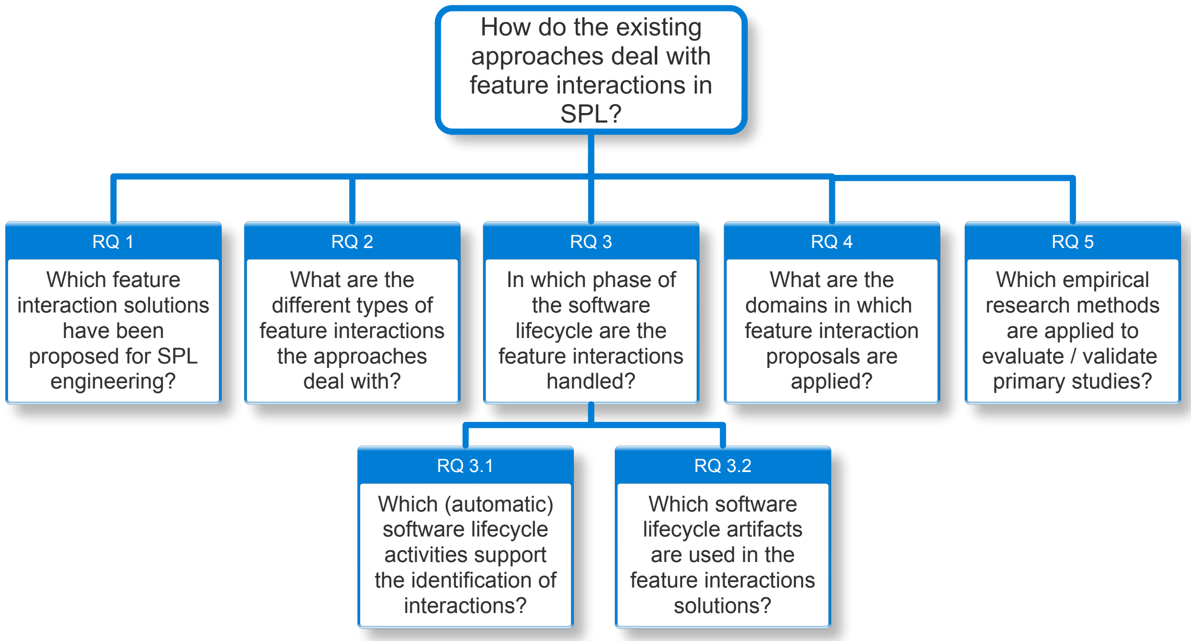

How do the existing approaches deal with feature interactions in SPL?
RQ 1
Which feature interaction solutions have been proposed for SPL engineering?
RQ 2
What are the different types of feature interactions the approaches deal with?
RQ 3
In which phase of the software lifecycle are the feature interactions handled?
RQ 4
What are the domains in which feature interaction proposals are applied?
RQ 5
Which empirical research methods are applied to evaluate / validate primary studies?
RQ 3.1
Which (automatic) software lifecycle activities support the identification of interactions?
RQ 3.2
Which software lifecycle artifacts are used in the feature interactions solutions?

Figure 3.2: Research Questions of the systematic mapping study

classification is related to the context of a feature interaction, i.e., interactions can be either externally-visible or internally-visible.

- *RQ 3. In which phase of the software lifecycle are the feature interactions handled?* FOSD is a paradigm that favors the systematic application of a feature in all phases of the software lifecycle [70]. FOSD aims at facilitating the structuration, reuse, and variation of software in a systematic and uniform way. In this question, we intend to identify in which lifecycle phase the feature interaction solution takes place. In addition, feature interaction solutions can be handled in many different activities of those phases, as for example, during the domain design phase or product configuration. In addition, the following sub-questions were derived:
  - *RQ 3.1. Which software lifecycle activities support the identification of interactions?* The process to identify and resolve interactions usually involves several activities in different software lifecycle phases. For example, the domain analysis phase may include domain scoping, feature modeling and automated reasoning; and the domain design and specification phase, may include architecture modeling and a formal specification [70]. Furthermore, in some cases, part of these activities are automated.
  - *RQ 3.2. Which software lifecycle artifacts are used in the feature interactions solutions?* Different artifacts can be used to support the feature interaction solutions during each phase of the software lifecycle. Examples of those artifacts are: requirements and feature model, project plan, business case and risk assessment.

Table 3.1: Digital Libraries

ACM Digital Library	http://dl.acm.org/
IEEE Xplore	http://ieeexplore.ieee.org/
Scopus	http://www.scopus.com/
Engineering Village	http://www.engineeringvillage.com/
Science Direct	http://www.sciencedirect.com/
Springer	http://www.springer.com/

Table 3.2: Search String

(”feature interaction” **OR** ”feature-interaction”)
AND
(”SPLE” **OR** ”SPL” **OR** ”product line” **OR** ”product family” **OR** ”domain engineering” **OR** ”application engineering” **OR** ”variant-rich” **OR** ”variability” **OR** ”feature-oriented” **OR** ”feature modeling” **OR** ”feature analysis” **OR** ”core asset”)
AND
(”software” **OR** ”system”)

- *RQ 4. What are the domains in which the feature interaction approaches are applied?* SPL can be applied in many different domains, such as, medical, financial and automotive embedded systems. Investigating the domains in which feature interactions approaches have been applied can indicate: (i) domains where the study on feature interactions are essential; and (ii) domains overlooking this research topic.

- *RQ 5. Which empirical research methods are commonly employed to assess the primary studies?* It is important to analyze the research rigor in feature interactions proposals for SPL engineering. In this question, we aim to investigate the applied empirical research methods (e.g. case study, controlled experiment, quasi-experiment and formal validation). It allows assessing the maturity of evaluation and validation in the area.

3.1.2 Search Strategy

The strategy of searching and selecting the primary studies consisted of three main phases: automatic search, manual search and full-text reading, as shown in Figure 3.3. For the first phase, a search query was executed in the digital libraries listed in Table 3.1. They are key publisher-specific resources [95] and cover almost all important conferences, journals and workshops research studies in the Software Engineering field. The automatic search covered all studies published up to the first quarter of 2016.

The search query was composed of a set of keywords defined according to the method of Kitchenham et al. [95]: (i) extracting software engineering concepts and terms from the research questions; (ii) reviewing terms used in the known papers; and (iii) identifying synonyms of the key terms. Based on that, three principal keywords were chosen: *feature interaction*, *product line* and *software*. They served as basis for building the full string, which was created with the addition of synonyms and alternative words, and joined with *AND* and *OR* operators, as Table 3.2 shows.

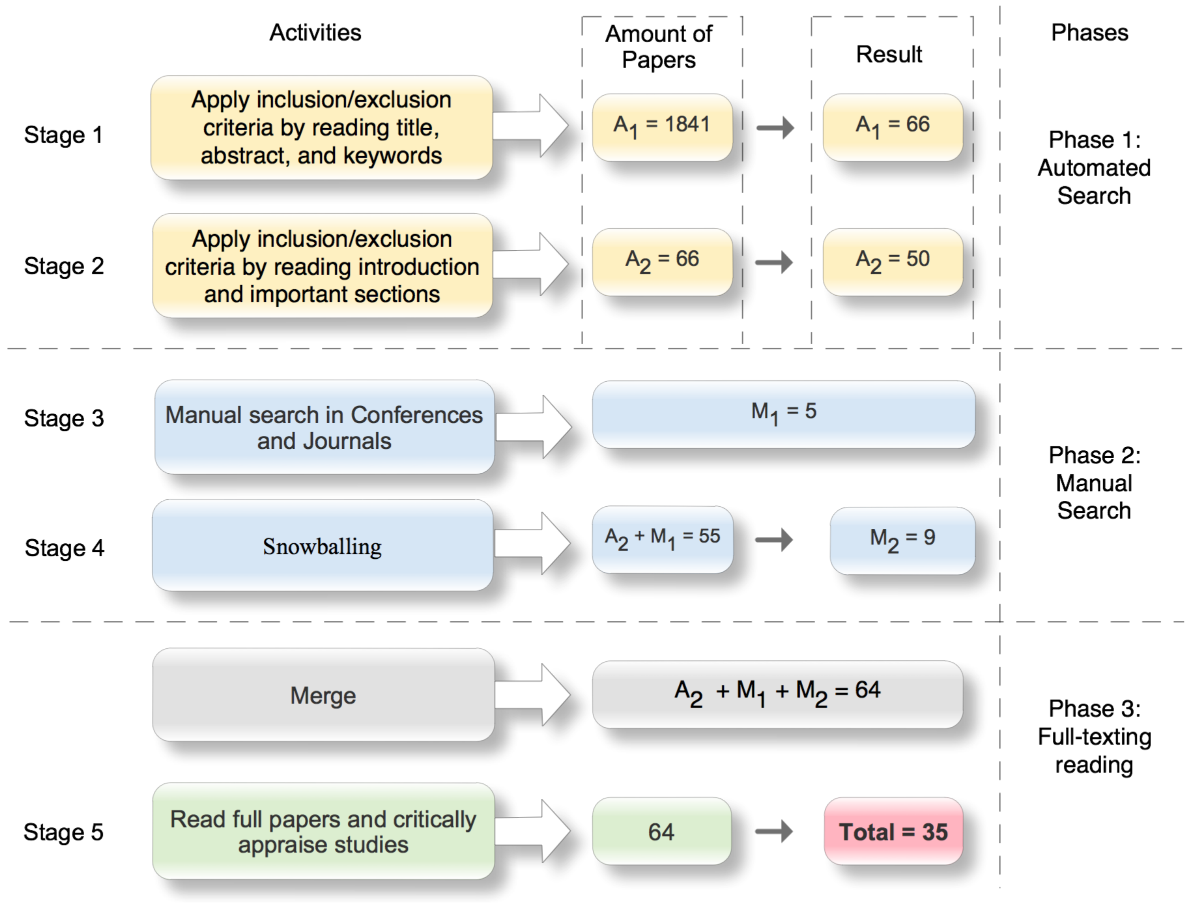

Activities
Amount of Papers
Result
Phases
Stage 1
Apply inclusion/exclusion criteria by reading title, abstract, and keywords
$A_1 = 1841$
$A_1 = 66$
Phase 1: Automated Search
Stage 2
Apply inclusion/exclusion criteria by reading introduction and important sections
$A_2 = 66$
$A_2 = 50$
Stage 3
Manual search in Conferences and Journals
$M_1 = 5$
Phase 2: Manual Search
Stage 4
Snowballing
$A_2 + M_1 = 55$
$M_2 = 9$
Merge
$A_2 + M_1 + M_2 = 64$
Phase 3: Full-texting reading
Stage 5
Read full papers and critically appraise studies
64
Total = 35

Figure 3.3: Search and selection process

After applying the search string in all search engines listed in Table 3.1, we collected a pool of 1841 studies. Figure 3.4 shows the share of papers per search engine, in which the IEEE Xplore and the Springer were the ones with more papers collected. We also collected 103 duplicated studies, since more than one digital library indexed the same venues. Most of the 1841 studies were not related to the topic of interest. To better select the studies and mainly eliminate those not directly related to the mapping study's goal, we defined a set of inclusion and exclusion criteria. Then, the automatic search was performed in two stages, as Figure 3.3 shows. In the first, the criteria were applied on the total of studies by reading their title, abstract and keywords. This stage excluded most of the non-related studies, resulting in a pool of 66 studies. In the second stage, we read the introduction and concluding remarks sections. From the set of 66 studies, we removed another 16 studies, based on the same set of inclusion and exclusion criteria, as follows:

- **Inclusion Criteria**
  - Written in English;
  - Peer-reviewed;

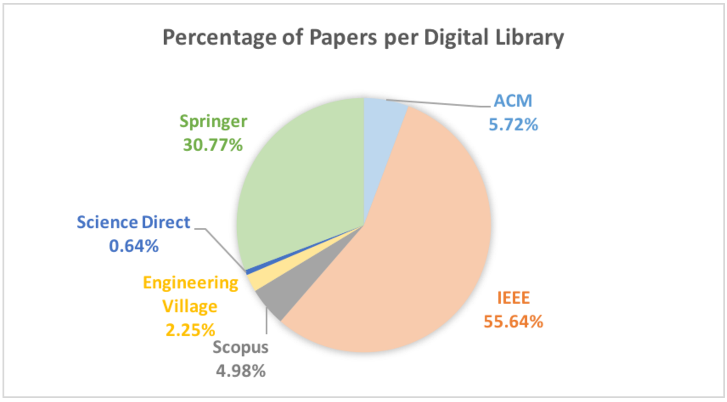


Figure 3.4: Percentage of papers collected in each Digital Library by using our search string

  - Addressing feature interaction in SPL engineering, in which a feature interaction occurs when the behavior of one feature is influenced by the presence of another feature.

- **Exclusion Criteria**

  - Short papers (less than 6 pages), studies describing tutorials, workshop and poster summaries, books;
  - Studies only related to single systems, i.e., when a study did not address feature interaction in SPL engineering;
  - Studies addressing feature interactions in a different meaning, such as dependencies (include and exclude constraints) and interactions between features in a feature model;
  - Studies that did not address any of the topics of the research questions;
  - Duplicate studies. When a study has been published in more than one venue, only the most complete version of the research is considered. The remaining studies are excluded.

The manual search also consisted of two activities: manual search in conferences proceedings and snowballing [95]. Previously, in the automated phase, we initially collected 1841 studies. Since that phase was executed in six well-recognized databases and we had already collected a significant amount of studies, the manual search was based on key conferences which commonly publish studies in the research area, which are: International Systems and Software Product Line Conference (SPLC), International Conference on Software Engineering (ICSE), Feature Interactions in Telecommunications and Software Systems (ICFI), and International Workshop on Feature-Oriented Software Development (FOSD Workshop). As a result, such a manual search yielded an additional set of 5 studies.

Snowballing consists of a manual search based on the reference list of relevant studies, and it is usually used to support the automated activity. This kind of reference search is also known as *backwards snowballing* [95]. For this activity, we analyzed the studies selected until this stage, i.e., 55 studies, 50 from the automated search and 5 from the manual search in conferences. The process consisted of analyzing the reference lists from those 55 studies, from which we selected 9. Thus, after merging the results from both automatic and manual search, we had a pool of 64 papers selected for the full-text reading phase.

In this last phase, we analyzed the remaining studies by carrying out a full-text reading and analysis. Then, we removed papers that: (i) presented an approach but it was not focused on feature interaction; (ii) presented a feature interaction discussion rather than a feature interaction solution; and (iii) had a more recent paper reporting the same approach. After performing a full-text reading, 35 primary studies were included in this systematic mapping study.

In order to ensure the reliability regarding the choice of the included studies, each study was evaluated by two researchers. Besides, they were responsible for designing the mapping protocol, searching candidate studies, reading and selecting the included studies, and also summarizing the results. Each accepted study underwent an agreement process, and in case of uncertainty and disagreement, the authors of the candidate study were contacted to solve and give appropriate guidance.

3.1.3 Update

We updated this systematic mapping study to cover papers published up to 2018. Previously, we had collected studies until the first quarter of 2016. Then, we conducted a new manual search and collected 5 more studies. We extended the preceding list of conferences, which included SPLC, ICSE, ICFI, and FOSD Workshop, and we also performed a manual search on the International Conference on Automated Software Engineering (ASE), European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS), International Conference on Generative Programming: Concepts Experiences (GPCE), and International Conference on Object- Oriented Programming, Systems, Languages, and Applications (OOPSLA).

After the updating, we now have a total of 40 primary studies. The next section presents our results including all studies selected up 2018.

3.2 RESULTS

The mapping process consisted of three phases: planning, conducting and documenting (Figure 3.1). As a result of the first and second phases, we selected 40 studies, as Table A.1 and Table A.2 in Appendix A shows. Figure 3.5 shows the distribution of the primary studies over time.

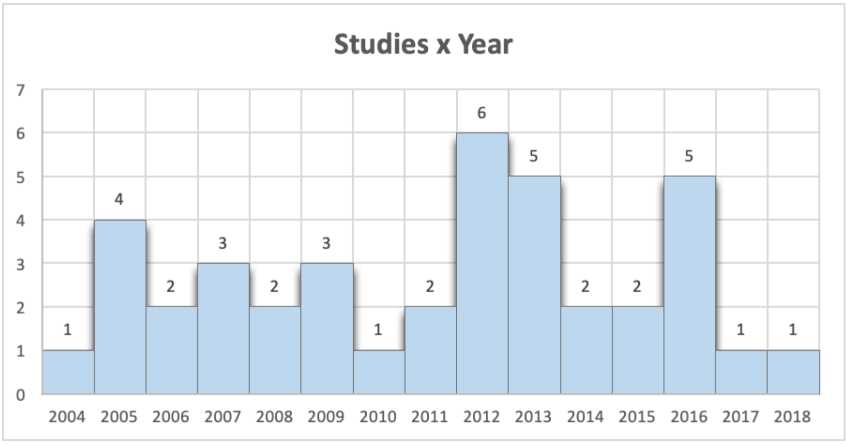


Figure 3.5: Number of studies by publication year

3.2.1 Classification Scheme

In order to analyze the studies, we first extracted the title, authors, venue and publication year. Next, we attempted to extract data to answer each of the research questions guided by a classification scheme, Furthermore, before proceeding with the data extraction, we performed a pilot study so that we authors could reach a mutual agreement in terms of research questions. This activity aimed at avoiding researcher's misinterpreted answers.

The scheme was based on the set of research questions designed for this study. The studies were published in journals, conferences and workshops from 2004 to 2018, and summarized according their venues in Table 3.3. Although we have not set a lower year-limit, the oldest paper was published in 2004. The selected studies were published in twenty-four different venues, from which two of them, namely the International Conference on Feature Interactions (ICFI) and the International Conference on Software Engineering (ICSE), stood out.

The classification scheme was developed iteratively and revised every time a new study was added. Each question classifies the study according to its own parameter, for example, RQ1 is related to solutions, RQ2 encompasses feature interactions types and RQ3 identifies the phase in the software lifecycle. Thus, as new papers were being analyzed, new categories within each question were created when needed. After reading the last selected study, the classification scheme was then established.

3.2.2 Feature Interaction Solutions

Different solutions have been proposed to handle feature interactions in different software development phases. In this study, we identified 3 main categories: detection, resolution and analysis.

The objective of a *detection* approach is to use or create strategies to identify cases of feature interaction. A *resolution* approach often considers that the interactions have already been identified, and focuses on solving the interaction problem to deliver the desired products. Finally, we named *analysis* approaches the rest, i.e., any approach that

Table 3.3: Studies' Venues

Venue	#
Int. Multi-Topic Conference	[9]
Int. Conf. and Workshops on Engineering of Computer-Based Systems	[28, 7]
Int. Conf. on Automated Software Engineering	[22, 96, 50]
Int. Conf. on Feature Interactions	[16, 97, 55, 14]
Int. Conf. on Fundamental Approaches to Software Engineering	[17]
Int. Conf. on Generative Programming	[21]
Int. Conf. on Model-Driven Engineering and Software Development	[98]
Int. Conf. on Software Composition	[30]
Int. Conf. on Software Engineering	[24, 11, 4, 25]
Int. Requirements Engineering Conference	[19]
Int. Software Product Line Conference	[26, 6, 99]
Int. Symposium on Software Reliability Engineering	[2]
Int. Symposium on the Foundations of Software Engineering	[20, 100, 101]
Int. Workshop on Feature-Oriented Software Development	[18, 29]
Int. Workshop on Variability Modelling of Software-intensive Systems	[54, 12]
Symp. on Foundations of Health Information Engineering and Sys.	[8]
Workshop on Model-Driven Engineering, Verification and Validation	[81]
FME Workshop on Formal Methods in Software Engineering	[13]
Journal of Software & Systems Modeling	[10]
Journal of Software Quality Journal	[5]
Journal of Automated Software Engineering	[23, 27]
Journal of Computer Networks	[3]
Journal of IET Communications	[15]
Journal of Information and Software Technology	[82]

aims to model, specify, manage and discuss feature interaction. These categories are not mutually exclusive, they can be combined to cover many software development lifecycle phases.

We classified the identified approaches according to the lifecycle phase when feature interaction is studied. We defined two main stages: *early phases* and *source code*. In the former, the studies deal with interactions without the need to work with source code; conversely, the latter considers studies that handle interactions using either the code of the system or the running system.

Table 3.4 shows the amount of studies by approach, as follows: (i) **Early Detection (De)**, when it takes place before the implementation, during feature modeling and specification activities; (ii) **Source Code Detection (Dsc)**, approaches that created/used a strategy to detect feature interactions in source code or used the runtime system; (iii) **Early Resolution (Re)**, solving interactions through specifications and models before the system implementation; (iv) **Source Code Resolution (Rsc)**, approaches that add, adapt or remove code to solve the interaction; (v) **Early Analysis (Ae)**, approaches that analyse feature interactions based on requirements, or other early artifacts; and finally (vi) **Source Code Analysis (Asc)**, which analyzes approaches (including interaction prevention) based on source code. We next discuss the categories of the identified approaches.

Table 3.4: Primary studies

Category	Primary studies
Early Detection (De)	[13, 15, 81, 7, 9, 16, 22, 17, 98, 11, 2, 18, 8, 5, 23, 96]
Source Code Detection (Dsc)	[12, 82, 25, 4, 6, 10, 3, 99, 100, 50]
Early Resolution (Re)	[54, 28, 98, 2, 20, 5, 27]
Source Code Resolution (Rsc)	[97, 30, 24, 26, 29, 10, 101]
Early Analysis (Ae)	[14, 55, 19]
Source Code Analysis (Asc)	[21]

3.2.2.1 Early Detection. Early Detection approaches concern about identifying feature interactions during the first stages of the software development lifecycle. Studies in this category aim to predict feature interactions based on models and specifications [15, 7, 98, 11, 18, 8, 5, 96]. For example, Atlee et al. [13] simulated violations among features based on *bisimilarity* in transition systems, and Hu et al. [15] proposed a Semantic Web-based policy interaction detection (SPIDER) method to automatically detect interactions from ontologies and *SWRL (Semantic Web Rule Language) rules.*

Aspect-oriented feature analysis techniques are also used to specify how features relate to each other, by providing separation of concerns at feature level [9, 98, 5]. These techniques predict interactions based on the identification of *dependencies* in crosscutting concerns aided by divergences either between dependency diagrams and other UML models [9] supported by critical pair analysis [98], or among goals described in a Goal-Oriented Requirement Language (GRL) [16]. In addition, Mussbacher et al. [5] used GRL combined with the Use Case Maps (UCM) notation to describe scenarios and architectures. GRL and UCM together constitute the User Requirements Notation (URN), which is extended with aspect-oriented concepts (AoURN). They aimed at identify interactions based on *traceability*, from feature model to goal model.

To support the identification and analysis of potential safety-critical feature interactions, Liu et al. [7] also proposed a traceability strategy, but in this case from requirements to fault models. Similarly, Bessling and Huhn [8] presented the System-Theoretic Process Analysis (STPA) to identify unexpected feature behavior and interactions of components through the specification of safety requirements, fault models and fault injection. Abdessalem et al. [96] also presented a model testing approach to respect system safety requirements. They introduced an automated approach to detect undesired feature interactions in self-driving systems at early stages.

Another class of studies uses *model checking* to detect interactions [22, 17, 11, 23]. The main idea is to generate appropriate interfaces on each feature to preserve its properties, and check for interactions by combining interface information. Usually, these approaches develop proof-of-concept prototypes based on propositional calculation [22], event calculus, SAT solver [17], and SAT-based symbolic model checking algorithms (IMC and IC3) [11], as a way to find violating products. The analysis of SPL requirements models allows early detection and correction of errors in features specification [11].

Early Detection approaches also exploit the *Alloy modeling language* to support the feature-oriented design. On the one hand, features are defined as Alloy modules and their correctness properties are specified in the post-conditions of the feature's actions, to be translated as Alloy assertions for consistency checking [2]. On the other hand, Dietrich et al. [81] first specify interactions with the feature-oriented requirements modeling language (FORML), then translating the FORML model into an Alloy model. In both approaches, the Alloy Analyzer is the responsible for the automatic feature interaction detection.

3.2.2.2 Source Code Detection (Dsc).

This category is related to approaches that detect feature interactions during or after the implementation phase. One strategy is to first annotate the source code with features specifications and then to use a *model checker* that automatically detect conflicts based on those specifications [4, 6, 3]. To accomplish the feature specification task, the following techniques have been used: design by contract [6, 3], AspectJ and FeatureAlloy [3], and assertions [4].

For example, Scholz et al. [6] used design by contract based on Java Modeling Language (JML) to formally specify the behavior of methods and classes, and a model checker to identify conflicts. Furthermore, Apel et al. [4] developed the SPLVerifier, a model-checking tool for C-based and Java-based SPL.

Other studies [12, 82, 25, 10] concern about the *variability aspects* of an SPL and how it can influence the feature interaction analysis. A common technique is the use of *ASTs (Abstract Syntax Trees)* to parse the source code to support the identification of feature interactions. Different parsers have been used, for example, TypeChef (C language) [82], Java Compiler Tree API [10], and Fuji tool (feature-oriented compiler for Java) [12].

Rodrigues et al. [82] analyzed the source code with TypeChef and counted, for example, the occurrence of *ifdef* statements and functions, as a way to compute dependencies among features. Linsbauer et al. [10] presented a variability extraction approach, based on ASTs, to identify traces and dependencies from features and feature interactions to their implementation artifacts. ASTs also supported the identification of feature-oriented design patterns, which enforce feature interactions by class refinements and caller-callee relations [12].

Features' combination may also lead to surprising *non-functional properties* results, i.e., when a particular combination of features has an unexpected influence on a non-functional property. For example, Siegmund et al. [25] carried out a performance prediction that determines the influence of an interaction on performance based on a small set of measurements on code; and Zhang et al. [99] proposed a mathematical model for performance measurement.

Lastly, other approaches aim to detect interactions based on the system execution. Nguyen et al. [100] and Meinicke et al. [50] proposed dynamic analysis tools for automatically discovering interactions. Nguyen et al. [100] presented iGen, which looks for interactions by running the system under a set of configurations to determine coverage information. iGen creates the minimal set of configurations that cover most of the interactions. Meinicke et al. [50] proposed a tool to execute all configurations simultaneously and allows users to inspect differences in data and control flow at runtime.

3.2.2.3 Early Resolution (Re).

The Early Resolution category consists in solving a problem caused by an interaction between two or more features through adaptations in models. Three studies that performed early detection (De) also presented mechanisms to resolve interactions [98, 5, 2]. Apel et al. [2] allow the developer to resolve interactions using Alloy derivatives, which *disable properties* of one of the involved features. In addition to disable properties, another strategy is to *change the order* in which the selected features are added in the product [98, 5].

Another set of *Re* approaches considers that feature interactions have been previously detected and then presents solutions to resolve interactions using modeling strategies [54, 28, 20, 27]. *Feature exclusion, mutual exclusion, dependencies, precedences and priorities* are examples of techniques to solve unintended interactions at specification time [20, 27]. Besides that, *ignoring interaction and feature adaptation* are also other examples [54]. For instance, Padmanabhan and Lutz [27] developed the DECIMAL tool and a decision model to investigate whether those techniques could be represented as constraints in DECIMAL. Moreover, Mosser et al. [54] considered the way an interaction is resolved as a variation point in the configuration process.

Feature model-based approaches [28] also represent a way to resolve interactions. For example, either a feature can be integrated to others or a new feature can be added in the feature model to resolve an interaction. In addition, Sochos et al. [28] proposed a new feature model relation, *interacts relation.* Unlike the previous ones, Bocovich and Atlee [20] presented a *fine grained method* to resolve interactions at requirements stage. For each feature, variables, actions and outputs are specified to respond system inputs and environmental conditions. Resolution strategies include features priority and to assign an output variable to either the average, minimum, maximum or the sum of the values described by the features' actions.

3.2.2.4 Source Code Resolution (Rsc).

Providing implementation alternatives is the main characteristic of source code-based feature interaction resolution approaches. Typically, those approaches explore solutions to the *optional feature problem* in the SPL development [30, 24, 26, 10], i.e., when optional features are apparently independent at their specifications, but are not in their implementation, indeed.

Hence, different resolution strategies have been proposed to deal with feature interactions, such as: *refactoring derivatives* [30, 24, 26, 10], *preprocessors* or similar tools for *conditional compilation* [97, 26, 29], *changing feature behavior*, *moving code* from one feature to another, *multiple implementations* per features, and resolution modules chosen at runtime [101]. Kästner et al. [26] discussed many of the aforementioned strategies to eliminate implementation dependencies and solve interactions related to the optional feature problem. However, those strategies may impair code quality, require additional effort, and decrease performance.

In this way, Takeyama and Chiba [30] proposed a design principle to reduce the effort in developing derivatives. They used a feature oriented programming language, the FeatureGluonJ (based on JAVA), to implement derivatives in a reusable manner for every combination of sub-features. Liu et al. [24] and Linsbauer et al. [10] also discussed derivatives as the main strategy to resolve interactions. Basically, the code that causes the interaction between features is removed from them and used to create a new module, separately.

Another common strategy is to use preprocessors by either annotating or colouring the features' source code. For example, to support the specification of features through algebra concepts, Batory et al. [29] painted each fragment of code in a program by at least one color. Silva et al. [97] presented a different approach, which consisted in annotating the source code with the Java annotations API, based on dependency models in both design and implementation stages.

Finally, the approach proposed by Zibaeenejad et al. [101] resolved conflicts among features at runtime. They implemented resolution modules that are chosen at runtime, depending on sensors outputs.

3.2.2.5 Early Analysis (Ae) and Source Code Analysis (Asc). Finally, the two last categories are related to approaches that do not discuss detections or resolutions of feature interactions, but rather propose ways to analyze them through either models and specifications based on early-phases artifacts (Ae) or source code (Asc).

Three studies were identified as Early Analysis (Ae) approaches [14, 55, 19]. Bredereke [14] and Gibson et al. [55] analyzed interactions based on the *requirements specification.* The former extended the formalism Z to specify a family of requirements into requirements modules, and used a type checker tool to point out contradictions in the family. The latter discussed possible interactions at the requirements stage during the analysis of an e-voting SPL system. Furthermore, a *modelling approach* was proposed by Shaker et al. [19] to explicitly model intended interactions among state-machine of features.

Unlike Ae approaches, Kim et al. [21] presented a source code-based analysis. This approach models interactions as a derivative tree and extended the CIDE tool, an Eclipse plugin for *coloring the source code*, to map different nestings of colors to that tree.

3.2.3 Software lifecycle

Figure 3.6 shows how the primary studies are spread over the FOSD phases: domain analysis (FDA), domain design & specification (FDD), domain implementation (FDI), and product configuration & generation (FPC). There are two main groups, early-stage approaches (De, Re and Ae) and source code approaches (Dsc, Rsc, Asc). The pattern presented in Figure 3.6 shows that FDA and FDD are more common in the former group, while FDI and FPC in the latter, as we discuss next.

Approaches focused on managing feature interactions at early stages, i.e., *early detection (De)*, *early resolution (Re)* and *early analysis (Ae)*, are mostly concentrated in working at FDA and FDD, as Figure 3.6 shows. Usually, FDD and FDA phases of early-stage approaches deal with: (i) (automatic) detection based on requirements modeling, features and software artifacts [13, 81, 9, 28, 98, 20, 18, 27, 96]; (ii) automatic reasoning [15, 98, 2]; (iii) formal specification based on model checking [22, 2, 23], SAT solvers [17, 11]; and also (iv) software safety analysis [7] with hazard definitions and failure modeling [8].

Meanwhile, as expected, only source code approaches (Dsc, Rsc and Asc) involved domain implementation (FDI) and product configuration & generation (FPC) phases. In general, they usually performed feature implementation and configuration activities

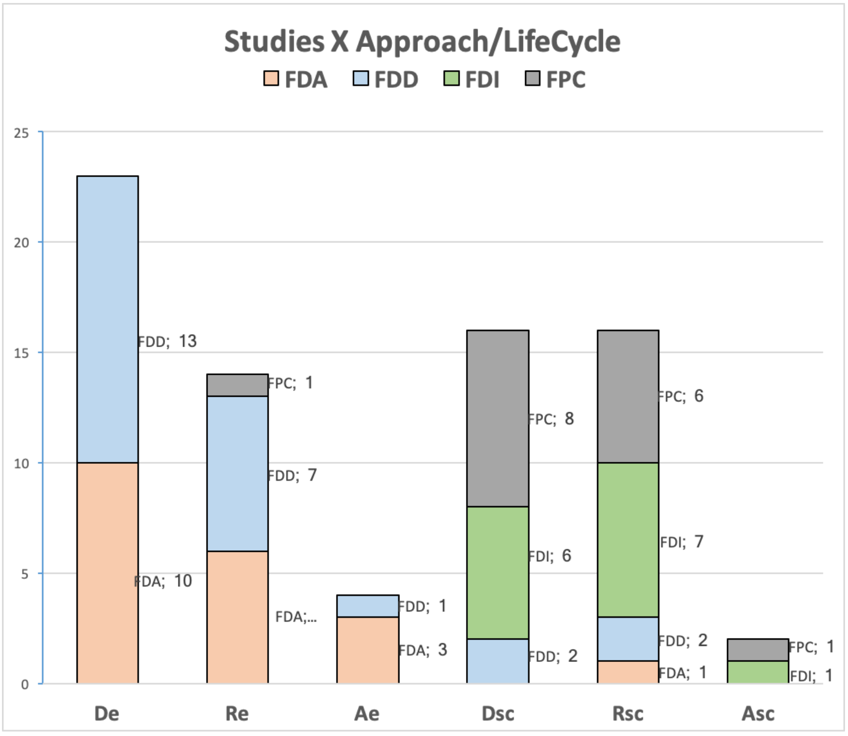


Figure 3.6: Lifecyle phases per approach. DA: Domain Analysis; DD: Domain Design and Specification; DI: Domain Implementation; and PC: Product Configuration and Generation

through: (i) feature modules, derivatives, feature selection, and composition [30, 24, 26, 6, 29, 10, 3, 101]; (ii) aspect specification [30, 4, 26, 3]; (iii) refactorings [12, 24]; (iii) source code annotations [97, 21, 6, 29]; (iv) preprocessor directives [82]; and (vi) the influence of performance on a product configuration [25, 99].

Although FDD and especially FDA are not the focus of source code approaches, two Dsc [6, 3] and two Rsc [97, 29] approaches presented activities in those phases, which consisted of strategies for either model checking approaches [6, 3], feature composition [29], or dependency models analysis in both design and implementation [97]. Furthermore, a group of those studies carried out partial automatic source code detection and resolution. To accomplish this task, they used different tools, such as SPL Conqueror [25], SPL Verifier [3], TypeChef [82] and Fuji tool [12].

3.2.3.1 Artifacts. Early and source code approaches adopted many software artifacts to support their activities. For example, source code approaches used classes, methods, fields, statements and variation points [12, 82, 29, 10]. Additionally, Table 3.5 shows various modeling techniques used to support feature interaction approaches.

Regarding the variability model, most approaches used the feature model as main model [12, 25, 9, 28, 30, 19, 18, 29, 10, 5], but often with slight differences and similar names, such as feature diagram [54, 17, 8], product model [8], feature machine [20], AoURN FM [5], feature structure tree [6], generic feature model [30] and variability

Table 3.5: Software artifacts

Category	Artifacts
UML diagrams	class, sequence, collaboration diagrams, state machine, use cases, activity models, business process
Goals	goal model, GRL model, environment model
Dependency	dependency model, dependency graph
Feature and product	feature model, feature diagram, product model, feature machine, AoURN FM, feature structure tree, generic feature model, variability model
Aspects	aspect feature model, point-cut feature model
Error/Fault	error model, fault injection model
Ontology	ontologies, world model
Graph	implication graph, call graph, AoURN's concern interaction graph, GRL graph, dependency graph

model [16].

Other models are very specific to their approach, as Table 3.5 shows, such as: (i) Goal Oriented Requirement Language (GRL) model [16]; (ii) aspect feature model and the point-cut feature model [9]; error model and fault injection model [8]; ontologies [15, 19]; dependency models [97, 21, 98, 10] and graphs [82, 25, 21, 4, 10, 5].

3.2.4 Feature interaction types

Unintended feature interactions are usually considered as either external or internal to the system [84]. Externally-visible interactions include functional and non-functional behaviors, and internal interactions involve structural and operational interactions. Figure 3.7 shows the amount of papers in each category.

Functional feature interactions are related to interactions that violate the functional specification of a configurable system. Since most approaches presented feature interactions solutions based on models and specifications, such as approaches not based on source code, they deal with interactions at the functional level, as Figure 3.7 shows.

However, functional interactions are also present in *Dsc* approaches [3, 4]. Apel et al. [3] analyzed whether feature-based specifications can be used to detect feature interactions in combination with formal specifications and verification techniques. Moreover, Apel et al. [4] provided a model-checking tool to check whether a feature composition satisfies the specifications of the involved features. Conversely, studies about non-functional feature interactions are less common than functional ones. For example, Siegmund et al. [25] proposed to predict system performance based on selected features. They aimed at detecting the system performance by analyzing its influence on the involved features. Zhang et al. [99] proposed a mathematical model that abstracts software systems to Boolean functions to identify performance-relevant feature interactions.

Although some source code approaches presented external interactions, most *Dsc*, *Rsc* and *Asc* approaches came up with solutions more related to internal feature interactions, i.e., structural and operational interactions. Studies about structural interactions were

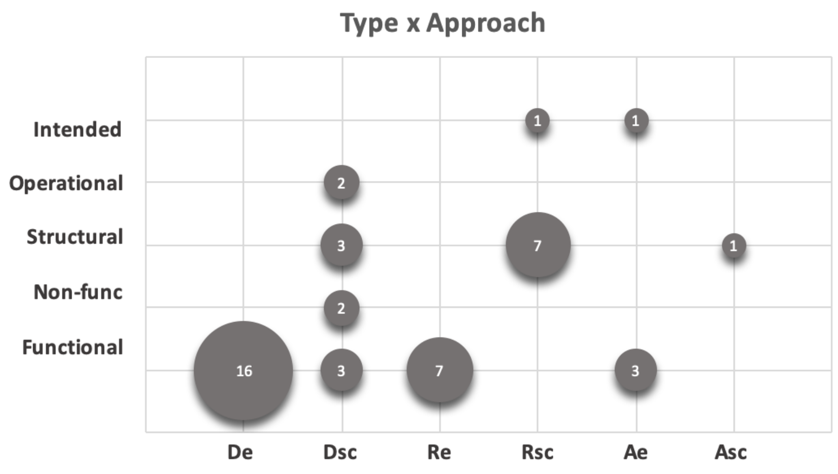


Figure 3.7: Number of approaches per feature interaction type. De: early detection; Dsc: source code detection; Re: early resolution; Rsc: source code resolution; Ae: early analysis: Asc: source code analysis

more frequent than operational ones, since the approaches usually work directly in the code through modules, directives and assertions.

Besides unintended interactions (functional, non-functional, structural and operational), Figure 3.7 shows two approaches that work strictly with intended interactions in SPL engineering [30, 19]. Takeyama and Chiba [30] discussed the difficulty to maintain a large number of derivatives, and Shaker et al. [19] proposed to model intended interactions as state-machine of features.

3.2.5 Domains

The 40 primary studies concern many application domains, such as smart homes, automotive, e-mail systems, database management systems, medical devices, phone systems, network systems and antivirus. Each primary study considers a different number of domains, from one up to forty program families with different purposes and sizes.

Figure 3.8 gives an overview of the distribution of the studies based on their application domain and feature interaction approach. The Figure only shows the domains that appeared more than once. For example, six different studies worked with *database systems* and four approached *automotive systems*.

Regarding the feature interaction approaches, we observed that some domains were more common in some approaches than in others. For instance, *Phone and telecommunication systems* were concentrated in *early approaches*, while *database systems*, *file compressor* and *games* were the opposite, they appeared in *source code approaches*, as Figure 3.8 shows. The predominance of De and Dsc in Figure 3.8, reflects the behavior of the set of primary studies, which has mostly detection approaches.

In addition to the domains showed in Figure 3.8, a large number of systems were also found. Among the types of systems that did not appear repeatedly in the set of primary studies are: text editor, web server, web browser, operating system, e-voting,

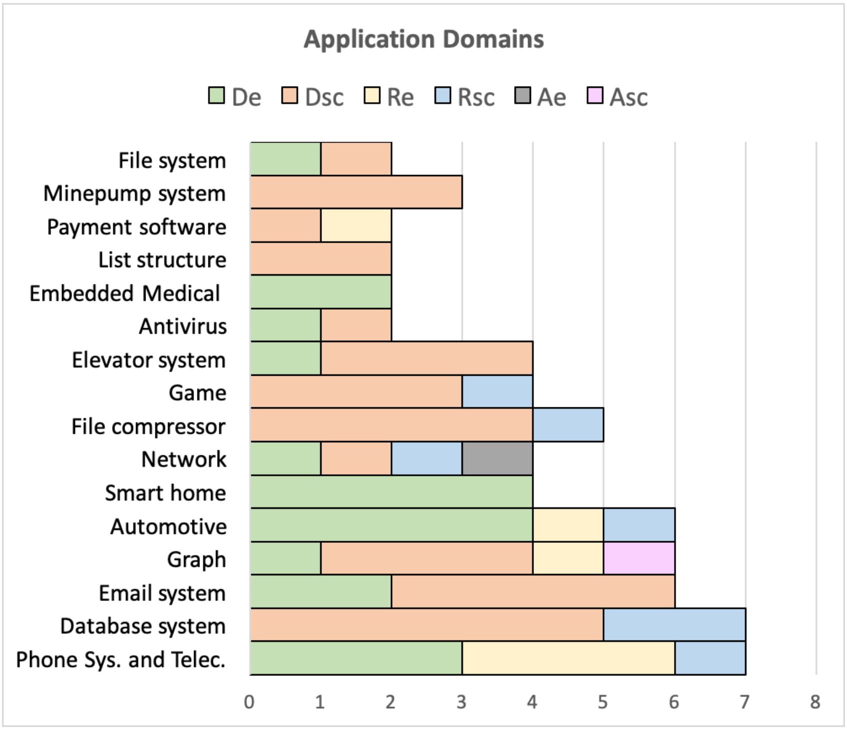


Figure 3.8: Number of studies by application domain. File System: [2, 3]; Minepump: [3, 4]; Payment: [5, 3]; List structure: [6, 3]; Embedded medical device: [7, 8]; Antivirus: [9, 10]; Elevator system: [11, 3, 4]; Games: [12, 13, 10]; File compressor: [13, 4, 10]; Network: [2, 3, 10, 14]; Smart home: [15, 16, 17, 18]; Automotive: [6, 19, 20, 11]; Graph: [2, 3, 21, 4, 12]; Email: [22, 23, 3, 4, 10]; Database: [24, 25, 26, 3, 12, 10]; Phone System and Telecommunications: [27, 28, 11, 19, 29, 30, 2].

bank system, microwave oven product line, graphical model editor, family of adaptation protocols, product-line that produce tools to manipulate Jak files, an event service SPL, and others.

3.2.6 Empirical assessment methods

From the set of included primary studies, 24 out of 40 employed *case study* as their empirical assessment method. However, only 5 presented a study detailed enough for an empirical research method, that is, employed a well-structured case study with basic information, such as hypothesis, research questions, methodology, results, discussion and threats to validity.

In the remainder subset, the central focus of their "case study" was to demonstrate the proposal in practice. The main used terms were: "explore the potential benefits and drawbacks of our approach", "demonstrate the utility", "demonstrate the approach", "illustrate our approach", "a demonstration of practicality and generality of our approach", and "case study as proof of concept". Other empirical assessment methods were also considered in the set of included primary studies, as Figure 3.9 shows.

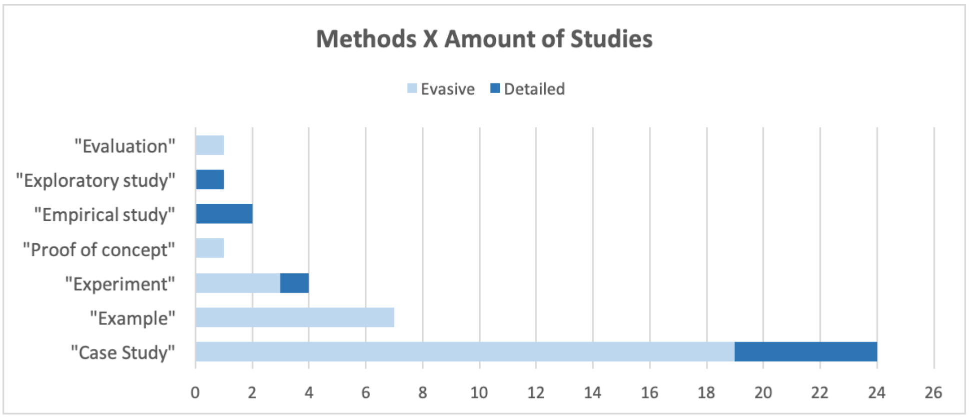


Figure 3.9: Empirical methods as named by the studies

Among the 40 studies, 9 presented an evaluation method with many details about the evaluation practices and process [12, 82, 25, 4, 10, 3, 100, 96, 50], as shown in Figure 3.9. For example, Rodrigues et al. [82] presented an *empirical study* to assess to what extent feature dependencies exist in actual software families following a well-designed process with goals, questions, metrics, subject selection, instrumentation, operation, results, discussion and threats to validity. They evaluated the source code of 40 software families from different domains.

In addition, Apel et al. [3] presented an *exploratory study* with 10 feature-oriented systems, and the *case studies* in [12, 25, 4, 10] evaluated their approaches with 6, 7, 6, 3 different systems, respectively. Apel et al. [4] evaluated its approach with 3 systems, but they implemented 2 versions (in C and Java) for each system. In addition, real-world case studies were performed in [12, 25, 10].

3.3 DISCUSSION

This section discusses the key findings of our study, as well as outlines directions for future research. In this systematic mapping study, we identified three main categories of approaches that handle detection, resolution and analysis of feature interaction. Each one was further classified according to the moment an interaction was managed, i.e., early phases studies or source code studies. Table A.1 and Table A.2 in Appendix A shows a summary of the 40 primary studies. Along this section, we discuss about the different feature interaction solutions, tools, domains and directions for further research.

3.3.1 Feature interaction solutions

When analyzing the selected studies, we could observe most of them consider either predicting interactions based on models or specifications, or solving and detecting interactions based on how the program code was implemented. Although some source code-based approaches also presented feature specifications [29, 3, 10] and modeling [54, 97, 6], they were focused on discussing strategies [6, 3] to identify interactions after implementing the software. For these approaches, source code represents the main software artifact.

We next present the main concerns about detection, resolution and analysis of feature interactions.

Detection approaches. Early detection (De) is the most represented category with 16 papers (Table 3.4). They predict feature interactions in SPL based on feature specification and modeling. A prior interaction detection may be used to support developers when implementing the features.

In this way, *De* approaches focused on identifying whether predefined constraints still remain after feature combinations. We identified seven different early detection strategies: (i) traceability from specifications to goal and fault models [7, 5]; (ii) dependencies in aspects cross-cutting concerns [9, 98]; (iii) dependencies in UML and goal models [16, 18]; (iv) verification of Alloy assertions [81, 2]; (v) model checking or propositional calculation of feature properties and models [22, 17, 11, 8, 23]; (vi) verification of SWRL inferences [102]; and (vii) bisimilarity in transition systems [13].

Conversely, source code approaches (Dsc) use models and specifications of features and software artifacts to analyze whether feature properties still remain after feature combinations. Those properties commonly arise from source code manual analysis. Others use code-coverage strategies based on dynamic analysis to find the locations where features interact. The *Dsc* approaches used four strategies to detect interactions: (i) model checker [4, 6, 3]; (ii) AST-based parser, [12, 82, 10]; (iii) non-functional properties measurements [25]; and (iv) runtime software [100, 50].

From the total of *Dsc* approaches, two of them do not use any kind of specifications to detect interactions [100, 50]. Instead, the programs need to inform which are the inputs that interact or the source code need to present feature annotations.[1] Nonetheless, Nguyen et al. [100] focused on code coverage and is only able to detect the localities in the code where features interact. Similarly, the tool and study presented by Meinicke et al. [50] identify where interactions occur based on control flow and data, but they focus on studying the essential configuration complexity. They show that the amount of variability that actually induces differences and interactions in the execution may be small enough to handle with a suitable analysis strategy.

Both, *De* and *Dsc*, have verification techniques based on model checking, the difference between them is that in the latter feature-based specifications are defined using both models and source code analysis. Typically, model checking strategies are combined with modeling (e.g.,design by contract) and implementation methods (e.g., code annotations, aspects). Another source code detection strategy is to use AST (Abstract Syntax Tree) to parse the code and support the identification of dependencies and design patterns.

De approaches usually do not depend on software programming languages and were mainly used to estimate feature interaction costs in software systems. Otherwise, *Dsc* approaches were concentrated on indicating either the absence or presence of feature interactions by means of specific languages: C, C++, and Java.

Resolution approaches. Early resolutions (Re) usually deal with model adaptations, especially regarding the arrangement of features. Five strategies were identified in SPL

[1]Feature annotations are annotations used to surround a piece of source code to show that piece correspond to a given feature.

Table 3.6: Papers and tools. C: category; A: assessment method.

Paper	Tool	C	A
[98]	MATA tool	De/Re	
[2]	FeatureAlloy	De/Re	
[5]	jUCMNav	De/Re	
[7]	DECIMAL, PLFaultCAT	De	
[17]	FIFramework	De	
[9]	Aspect miner tool	De	
[81]	FORML2Alloy	De	
[8]	SCADE, VIATRA	De	
[11]	ABC toolset	De	
[28]	DOME tool, IBM's Rat. Req. Pro, RPM Package Manager	Re	
[27]	DECIMAL tool	Re	
[14]	type checker tool, CADiZ	Ae	
[6]	SpeK, FeatureHouse, ESC/Java2 and Simplify	Dsc	
[25]	SPL Conqueror	Dsc	✓
[3]	FeatureHouse	Dsc	✓
[4]	SPLVerifier, FeatureHouse, CPAChecker, JAVA PathFinder	Dsc	✓
[82]	TypeChef	Dsc	✓
[100]	iGen	Dsc/runtime	✓
[50]	VarexJ	Dsc/runtime	✓
[24]	AHEAD Tool Suite	Rsc	
[21]	modified CIDE, FMCA	Asc	

approaches: (i) disabling properties inside a feature [2]; (ii) change feature order [98, 5]; (iii) feature exclusion, precedence and adaptation [54, 27]; (iv) addition of features and relations, feature composition [28]; and (v) feature prioritization [54, 20]. These strategies are usually supported with alloy specifications, goal, aspects, decision, UML, and dependency models.

Source code resolutions (Rsc), as the name says, propose fixing interactions problems on code. We also identified five strategies, as follows: (i) feature behavior changes [26]; (ii) moving code between features [26]; (iii) multiple feature implementations [26]; (iv) conditional compilation [26], coloring code [29], annotations [97]; and (v) refactoring derivatives [30, 24, 26, 10].

In addition, four out of thirteen resolution approaches also presented how to detect interactions. The remainder assumed that the interactions had already been identified and did not provide details of the identification process. Regarding the programming language, similarly to *Dsc*, the *Rsc* approaches worked with C, C++, and Java. The latter was the most common programming language used in both *Dsc* and *Rsc* approaches.

Analysis approaches. Finally, early analysis (Ae) and source code analysis (Asc) represented the category of studies that neither detected nor resolved interactions, but presented ways to improve modeling [21, 19], requirements specification [14, 55] and source code derivatives [21]. The latter discussed derivatives with AHEAD and AST.

3.3.2 Tools and validation

Most approaches predict interactions based on the behavior of features and products expressed in a set of states, transitions [13, 81, 7, 98, 11, 8] and model checking [22, 17, 2, 23]. Many of them are supported by tools that treat features as formal expressions, such as model checker-based tools [81, 2] and SAT solvers [17, 11]; critical pair analysis tool [98], or even specific requirements design, analysis tools [7, 8] and aspect-based tools [9, 5].

In addition, *Dsc* and *Rsc* approaches are mainly dependent on implementation alternatives to find solutions for feature interaction problems, such as design patterns [12], feature modules and derivatives [30, 24, 26, 29, 3, 101], and code annotations [4, 29, 10]. Five out of ten *Dsc* approaches and one *Rsc* were assisted by tools to support the implementation process, as for example: FeatureHouse[2], AHEAD Tool Suite[3], CIDE[4] and SPL Conqueror[5].

Table 3.6 shows the primary studies, corresponding tools, and feature interaction solution category. In general, 60% of early-stage approaches (*De*, *Re* and *Ae*) were supported by tools [14, 81, 7, 9, 28, 17, 98, 11, 2, 8, 5, 27] and 50% of source code papers (*Dsc*, *Rsc* and *Asc*) presented tools to assist their approach [82, 25, 21, 24, 4, 6, 3].

Although some approaches were supported by tools, only a few of them presented detailed empirical assessment methods [25, 3, 4, 82], as Table 3.6 shows. Actually, detailed methods were only presented in source code approaches, usually through a mix of real and toy SPL. For example, Apel et al. [3] conducted an exploratory study on the basis of ten small JAVA systems with existing specifications. Conversely, Rodrigues et al. [82] analyzed 40 industrial families written in C.

Early-stage approaches (De, Re, and Ae), even those ones supported by tools, presented evasive experimental results. Most of them aimed to exemplify the approach or present a proof of concept. In general, only 17% of the total studies conducted a proper and detailed validation. It is noticed that performing empirical studies is still challenging in the software systems field.

3.3.3 Domains

The feature interaction problem has been discussed in the telecommunications domain for years. During the 80's, the concerns of the telecommunications community revolved around the future. The community was looking for a fast introduction of new features in the systems as well as a smooth separation of applications from the system. This is currently referred to as modularization, components development, or even feature-oriented development.

Figure 3.10 shows the four most common domains regarding the paper's publication year. The study of feature interaction in *phone systems and telecommunications* is still present in recent years, as well as other domains, such as databases, email, and graph

[2]http://www.infosun.fim.uni-passau.de/spl/apel/fh/
[3]https://www.cs.utexas.edu/users/schwartz/ATS.html
[4]http://wwwiti.cs.uni-magdeburg.de/iti_db/research/cide/
[5]http://www.infosun.fim.uni-passau.de/se/projects/splconqueror/

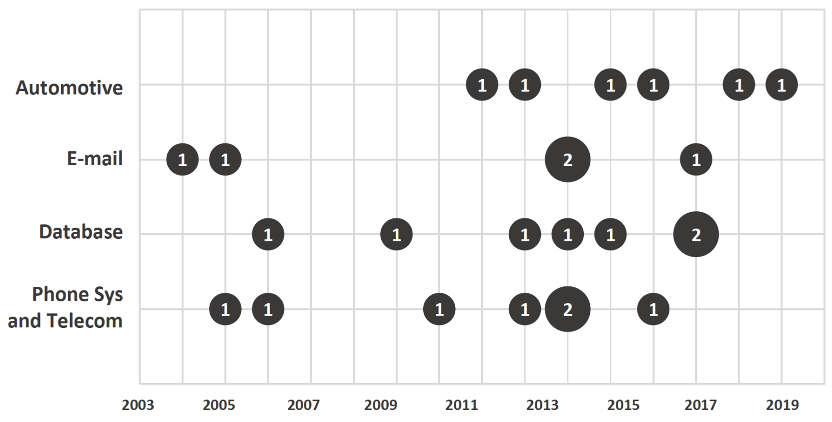


Figure 3.10: Domains over time.

systems. Furthermore, Figure 3.11 presents the most common domains addressed by research, from three perspectives: (i) approaches, domains, and amount of studies; (ii) approaches, FOSD phases, and amount of studies; and (iii) approaches, domains, and FOSD phases. Domains spread over both early and source code approaches have broader coverage in relation to software lifecycle phases, compared to approaches concentrated on only either early or source code stage.

On the one hand, telecommunication and phone [28, 30, 11, 19, 2, 29, 27], email [22, 4, 10, 23, 3], graph [12, 21, 4, 2, 3], and network [14, 2, 10, 3] systems involved activities in all areas of the lifecycle, such as: (i) FDA: requirements analysis, feature modeling, automatic reasoning; (ii) FDD: formal specification, model checking, Alloy; (iii) FDI: implementation techniques, modules, aspects, design patterns; (iv) FPC: composition, feature selection. On the other hand, automotive systems did not involve FDI activities, as well as smart homes, which did not consider neither FDI nor FPC activities. Due to difficulties to create real test cases to automotive systems and smart homes, feature interaction approaches usually identify interactions through simulations based on models and specifications.

Despite early detection (De) having twice as many studies compared to source code detection (Dsc), both categories are the ones with the most variety of domains. Although *Dsc* has fewer papers, they deal with a greater number of different domains in their approaches. While 12 out of 16 *De* papers discussed their approaches in 1 domain, almost all *Dsc* papers (except 1) worked with at least 6 domains. For example, Rodrigues et al. [82] performed an empirical study covering 37 domains with different industrial software families.

Conversely, *De* papers are usually tied to their own field of study. For example, Hu et al. [15] proposed an ontology-based approach that is specific to the smart home domain. A discussion on whether it is applicable to a different domain is missing. The same occurs for [55, 7, 16, 18, 8, 5]. Accordingly, the modeling and specification approaches of the *De* category are usually more specific and may be less generalizable than the *Dsc* approaches.

Even when the authors claimed their approaches could be generalized to other areas,

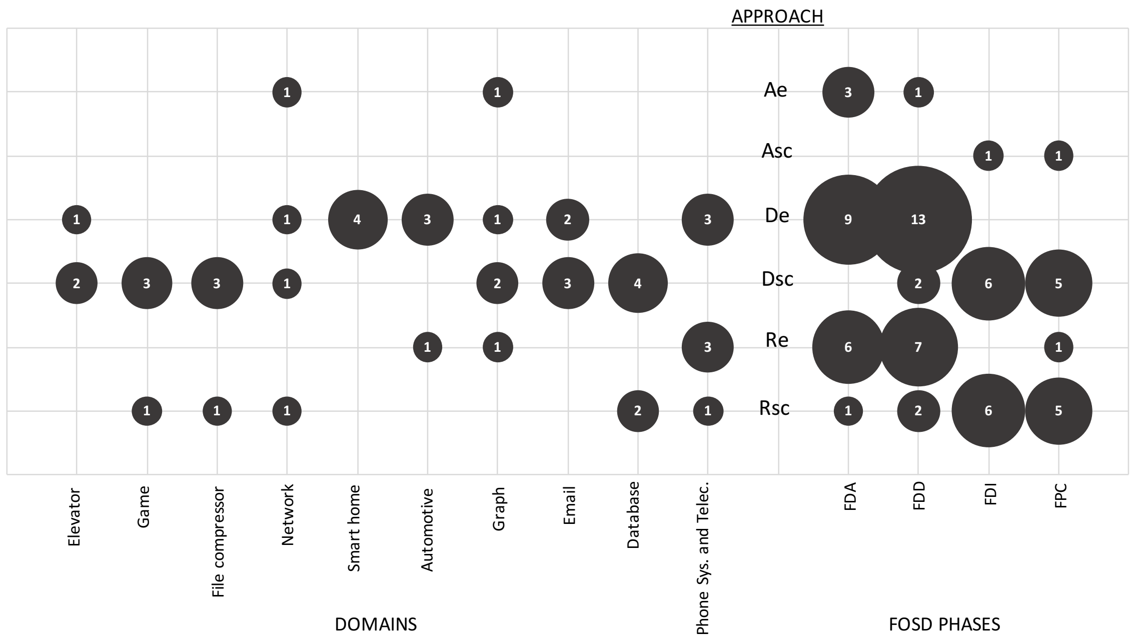


Figure 3.11: Approaches, Domains, and FOSD phases

the studies do not discuss how relevant the results would be for other contexts; there is no discussion about the implication of the results achieved for other application domains, nor any discussion about possible and potential limitations of using the proposed approaches, which limits any inference in this sense.

3.4 THREATS TO VALIDITY

There are some threats to the validity of our study. They are discussed next:

- **Internal Validity.** The literature search was in part guided by keywords. This is not ideal for our topic, since the words used for describing it are used in other fields with a very different meaning. This is reflected by the large number of irrelevant responses to our queries, that has been carefully sorted manually. Conversely, it is possible that some studies of a related topic in a different field use a different vocabulary, and would thus have been missed. We used snowballing to recover some references that could have been missed by the keyword approach.

- **Construct validity.** Research questions may represent a threat in systematic studies. In out case, our research questions are relatively general, since the goal of this study is to give an overview of the field. There may be a set of more specialized research questions to explore in further research. Another threat is related to the decision of which studies to include/exclude. To minimize this threat the processes of inclusion and exclusion were piloted by two researches.

  The papers have been classified by one researcher mainly, with the other researchers verifying the adequacy. This threat is known as subjectivity bias. However, we contacted the papers authors in cases of disagreements or questions. The procedure performed in this study has a compromise in terms of effort and objectivity.

- **External validity.** Publication bias is a common threat. Our study is limited to approaches published in the scientific literature. There may be another set of approaches, as important as those analyzed in this investigation, which has not been published as regular research papers, in particular when developed inside companies that cannot be published for strategic reasons. Another bias is the restriction to publications in English. Since the field uses mainly English, the introduced bias is minimal.

3.5 ADDRESSED GAP AND DIRECTIONS FOR FURTHER RESEARCH

Early detection (*De*) has over twice as many studies compared to source code detection (*Dsc*). In addition, *De* approaches may not be able to identify all concrete interactions, because many of them can only be detected with the software source code. *De* approaches are basically based on models. Although *Dsc* approaches are commonly reported in the literature, they present many disadvantages, such as the large amount of interactions that can be detected, which may include many false positives.

As discussed in Section 3.3.1, detection approaches (both *De* and *Dsc*) are mostly specification-dependent. They usually require an expressive number of specifications and variant combinations, which are usually defined manually. Commonly, they need a costly and formal specification of features, products or dependencies. However, software specifications are usually either missing or incomplete in the software industry.

The approaches that do not use specifications to detect interactions are based on the running software. Nguyen et al. [100] and Meinicke et al. [50] proposed dynamic approaches to detect feature interactions. Both studies identify where features interact, but they do not provide a way to distinguish from the set of interactions which ones are problematic to the system. They just provide the total of possible interactions, without any clue to which interactions may represent a risk to the program's behavior. Features in software systems are likely to interact many times and in many different ways. Showing the places where all features interact do not provide sufficient insights to support developers to find and fix problems. In addition, since those approaches focus on other aspects (code coverage [100] and configuration complexity [50]), they present much more information that is not related to the interactions of features. Thus, it gets hard to understand which features are involved in the interactions and how to fix the problems.

Often, detection strategies are partial and only address specific points of a feature interaction investigation. On the one hand, when interactions are detected, it is not possible to identify which interactions are causing problems. On the other hand, if they are resolved with a specialized module or implementation changes, the previous step on how they came up with those interactions is not explained. Particularly, those work do not explain how they distinguish the desired interactions from the problematic ones. In our investigation, we observed that most of the approaches provide limited experimental results, rarely perform effective and detailed case studies.

Based on the yielded results from this mapping study, we propose a dynamic analysis of systems able to: (i) identify interactions without depending on software specifications; (ii) present a friendly, clean, and interactive interface to show which features interact for each interaction; and (iii) support developers to distinguish desired (benign) interactions from the problematic ones; Tool support could also assist developers to implement features by informing at runtime which types of interactions they have, and the specific conditions that interactions are triggered, such as inputs and variables. Along this idea, we aim to analyze interactions from the running software and provide insights to the developers to help them identify interactions that lead to a bug. We present the details of our approach on next chapters.

3.6 CHAPTER SUMMARY

In this chapter, we presented a systematic mapping study to investigate the state-of-the-art of feature interactions in SPL engineering. The 40 included primary studies were classified according to their proposed solutions (RQ1), feature interactions types (RQ2), software lifecycle (RQ3), software domains (RQ4) and empirical assessment methods (RQ5).

More than 60% of the studies either provided an initial discussion about feature interaction management or discussed how to identify interactions at early phases of the SPL development, mainly based on traceability, dependencies, verification of Alloy assertions, feature exclusion, precedence, and adaptation. In addition, the set of existing approaches is strictly dependent on software specifications and generally provide limited experimental results, rarely perform effective and detailed case studies.

Next chapter presents our dynamic approach to identify interactions. We aim to overcome some of the drawbacks identified in the detection approaches, such as: we do not need specifications to detect interactions, and we detect real interactions which come from the software execution.

PART III

A DYNAMIC ANALYSIS APPROACH WITH VARXPLORER

Chapter

4

ON THE DETECTION OF FEATURE INTERACTIONS

The goal of this chapter is to introduce the process responsible for dynamically detecting interactions in highly configurable systems, which is based on *variational execution*. We provide an inspection process that provides developers with an easier means to distinguish intended interactions from interactions that may lead to bugs.

Thus, to detect feature interactions in a test execution (without knowing whether they are benign), we use the variational interpreter VarexJ [49, 50]. It performs variational execution to simultaneously execute all system configurations. VarexJ reveals the differences among configurations on both control and data flow [50].

In this chapter, we introduce the basic concepts of multiple executions and variability-aware execution. Thus, the chapter consists of six main sections, as follows:

Section 4.1 presents the motivation of this detection process;

Section 4.2 shows a running example;

Section 4.3 discusses how specifications can support the detection of interactions;

Section 4.4 presents the different strategies to detect interactions;

Section 4.5 shows details about the dynamic approach we use to find the interactions, which is based on variational execution;

Section 4.6 finally presents VarexJ, the variational interpreter we use to run the systems;

4.1 WHY SHOULD WE DETECT FEATURE INTERACTIONS?

Often, the software development team needs to deal with unexpected system behavior due to interactions between features. Features developed and tested separately may present a different behavior when combined in a system [84]. A *feature interaction* is observed when the combined behavior of two or more features differs from the individual behaviors of both features [103]. For example, one feature can interfere with, enable, or overwrite the effects of another feature.

Features are frequently combined to cooperate and contribute to an intended behavior (expected interactions). However, most interactions cannot be predicted upfront. Unexpected interactions can be classified as either benign or problematic to a system. Problematic are undesired feature interactions that may cause faulty or damaging system behavior, such as crashes. However, most interactions, although unexpected, may result in a benign behavior that does not cause any problem to the system or might even be essential to coordinate the functionalities of multiple features. Identifying and classifying feature interactions is challenging as they only appear in certain test cases and configurations (variants of a system composed of different feature combinations).

4.2 RUNNING EXAMPLE: WORDPRESS

Listing 4.1 shows an example illustrating both benign and problematic behavior. In the code excerpt modeled after WordPress[1], an extendable blogging and content management system, the features *weather* and *fahrenheit* interact intentionally to display the weather information in a desired format. However, the feature *smiley* interacts with the feature *weather* in an unintended way, although they do not crash the system. When they are together in the same system, the temperature is not showed and the system presents an unexpected output. In some executions, as *smiley* replaces a part of the weather tag, the feature *weather* has no effect if *smiley* is selected. Thus, instead of the "[:weather:]" tag be replaced by the current weather (e.g., 70°F), it is rewritten to "[:weather☺" and presented to the user in place of the temperature.

Figure 4.1 shows the output of two configurations. For *Configuration 1* (*weather-fahrenheit*), the current temperature is shown to the user, indeed. However, for *Configuration 2* (*weather-smiley*), a HTML tag is shown and the user cannot see the temperature.

4.3 FEATURE-BASED SPECIFICATIONS AND GLOBAL SPECIFICATIONS

Detecting all feature interactions requires having specifications for all system configurations. However, this usually does not scale to a large number of possible configurations. Another strategy is to specify features in isolation; a *feature-based specification* describes the behavior of a feature in isolation without any explicit reference to other features [37]. Such behavior is supposed to be preserved independent on other features in the system. For example, a feature-based specification for the feature *Smiley* (Listing 4.1) is that a specific composition of characters in the HTML must be changed by a smiley image[2]:

$$\mathbf{AG}\ \mathit{createHTML}(c) \Rightarrow \mathit{c.contains}(\text{“ :]”})\ \mathbf{R}\ \mathit{c.contains}(\text{“}\mathit{smiley.png}\text{”})$$

[1] ¡https://developer.wordpress.org/¿

[2] AG states that the proposition must hold globally for all execution paths. The operator R states that the proposition to the left must hold until (and including) the state described by the proposition to the right has been reached [3].

Listing 4.1: Simplified WordPress example [50].

```

boolean STATISTICS, SMILEY, WEATHER, FAHRENHEIT, SECURE_LOGIN;

    void createHtml(String c) {
        c = wpGetContent();
        if (SMILEY)
            c = c.replace(":]", getSmiley(":]"));
        if (WEATHER) {
            String weather = getWeather();
            c = c.replace("[:weather:]", weather);
        }
        String head = initHeader();
        print("<html><head>" + head + "</head><body>");
        if (STATISTICS) {
            int time = getCurrentTime();
            printStatistics(time);
        }
        print("<div>" + c + "</div>");
        String foot = wpGenFooter();
        print("<hr/>" + foot + "</body></html>");
    }

    String getWeather() {
        float temparature = 30;
        if (FAHRENHEIT)
            return (temperature * 1.8 + 32) + "°F";
        return temperature + "°C";
     }
```

Several approaches work with feature-based specifications to detect interactions. Li et. al [44] present a model checking approach to detect interactions automatically given a group of feature specifications. The approach tests CTL (computation tree logic) properties of features to identify cases in which the specification is violated. Apel et al. [45] also propose a technique to verify whether specifications hold across system configurations. To perform this verification, specifications for intended interactions may be needed, and each feature requires a formal specification of its behavior.

With feature-based specifications, interaction faults can be detected when a feature specification is violated in a configuration. In practice, nevertheless, it is uncommon to create specifications for all features. In general, approaches based on feature specifications present two main drawbacks: (1) from the whole set of features, it is not clear which combinations of features need to be verified and (2) verification tools need precise specifications to check against, information that developers are often reluctant to prepare or unable to accomplish.

Conversely, *global specifications* represent a widespread strategy to reduce the effort of creating specifications for individual features or specific program variants, since they cover all configurations using general requirements [37]. Typical global specifications are requirements that the system should not crash and that fulfills certain functional requirements in all configurations (e.g., passes all test cases). For the WordPress example, a global specification is that the HTML page must be correctly created and showed to the user:

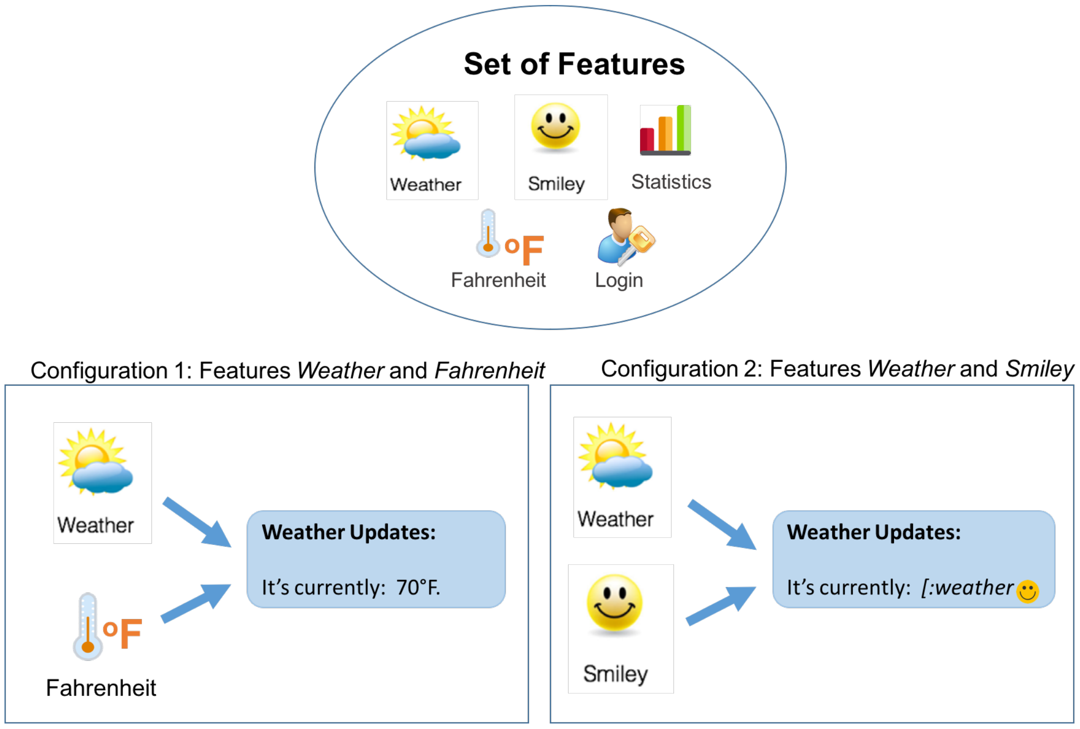


Figure 4.1: Example of two Wordpress configurations.

$$\mathbf{AG}\ createHTML(c) \Rightarrow isValidHTML(c)$$

Global specifications only describe properties for all system configurations, and thus cannot describe nuances of intended and unintended interactions to recognize if they affect feature behavior. Generally, it is difficult to find bugs caused by unintended interactions without any specification. In this way, despite their disadvantages, global specifications provide a convenient way of detecting interactions. For that reason, many studies base their approaches on that kind of specifications and focus on exploring the configuration space [41, 49, 50, 51, 52, 46, 38].

4.4 STRATEGIES TO DETECT INTERACTIONS

Highly configurable systems can be composed from a set of thousands of features (aka. configuration options) [104]. For example, Eclipse has more than 1,600 plugins [31] and the Linux kernel has more than 15,000 configurable options [32, 33]. This large set of options may be combined in different ways, and developers should guarantee that all valid combinations work properly.

Usually, a test may pass to a given configuration and inputs, but it fails if a parameter is changed [41, 42]. Feature interaction bugs are hard to be identified, especially those that do not lead the system to a crash, but cause a wrong behaviour. To detect them, we would need to compare the executions of all system configurations. Feature interactions are then manifested in the differences in data and in the control flow that depend on multiple features. However, executing one configuration at a time does not scale for large and real software. The number of possible configurations can potentially be exponential to the number of options/features.

Recent analysis focus on detecting feature interaction bugs from global specifications, in which all configurations of a configurable system need to fulfill. Usually, these approaches check global specifications based on systematic sampling [38, 39, 40], model checking [44, 46, 47, 48], combinatorial interaction testing [41, 42, 43], and variational execution [49, 50, 51, 52].

Sampling approaches are able to detect configurations that fail, but do not detect unexpected and undesired behaviors. Bugs that do not cause crashes might be as critical as the ones that lead the system to fail. Besides that, sampling strategies usually do not detect the interaction that causes the bug. In addition, static analysis [105, 106, 107] identify interactions based on estimated values, only. Although they do not need to use program inputs as dynamic analysis do, static analysis tend to over approximated potential interactions. Other studies execute configurations separately and use symbolic execution as a strategy to identify interaction problems. For example, Reisner et al. [108], measured the effect of interactions only on control flow, whereas variational execution analyzes both control and data flow.

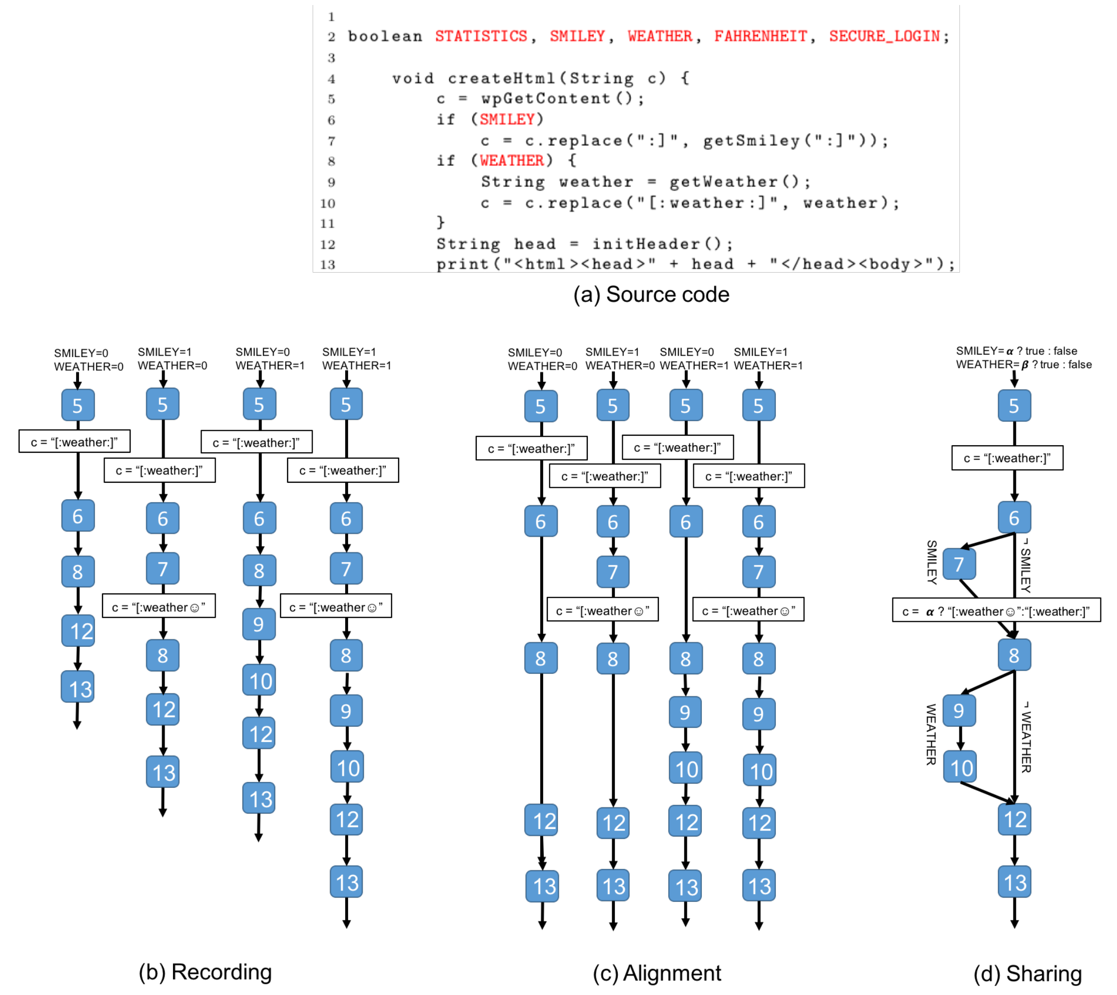
There is a lot of work to detect faults caused by feature interactions, as well as techniques to resolve them. However, since specifications at the feature level are usually missing, the above mentioned approaches may not detect all incorrect system behavior, especially bugs not covered by global specifications and bugs that do not result in a crash or other easily observable behavior. Furthermore, variational execution: (i) provides dynamic analysis of the software; (ii) presents control and data flow interactions; and (iii) provides concrete variable values to each possible configuration.

4.5 VARIATIONAL EXECUTION

Variational execution approaches provide multi-execution to synchronize similar concrete executions [50]. We can run the program with multiple inputs to understand how the differences affect program behavior. The main goal is to execute a test case exhaustively over all configurations of a software product (i.e., all combinations of features or inputs). Since many executions of a system are similar [49], we do not need to run all the possible configurations in separate.

For variability-aware execution, we require an interpreter as basis. A variability-aware interpreter is a technique that uses multi execution to overcome the exponential explosion of feature combinations. In practice, it runs the code that is common just once. When it finds a configuration option, it opens multiple branches. Therefore, for the parts of the program that are only executed under specific conditions, the interpreter will execute these parts only under these conditions. After executing those parts that are different, and for the next step, different software configurations may have the same instructions to be executed. A variability-aware interpreter attempts to continue the execution of the rest of the code again only once if possible.

In general, variational execution runs all program configurations, often efficiently, by



```
1
2  boolean STATISTICS, SMILEY, WEATHER, FAHRENHEIT, SECURE_LOGIN;
3
4      void createHtml(String c) {
5          c = wpGetContent();
6          if (SMILEY)
7              c = c.replace(":]", getSmiley(":]"));
8          if (WEATHER) {
9              String weather = getWeather();
10             c = c.replace("[:weather:]", weather);
11         }
12         String head = initHeader();
13         print("<html><head>" + head + "</head><body>");
```

(a) Source code

(b) Recording (c) Alignment (d) Sharing

Figure 4.2: WordPress example being executed with a traditional approach (*recording*) versus the variational execution (*sharing*). This code focus on two options: smiley and weather (the number on the elements indicate the line number).

sharing redundancies of the executions and values among these configurations [50]. The main idea is to split the execution when the code presents differences and to join again when it has to execute the same code.

Figure 4.2 shows how this process works for the WordPress example. In order to make it simpler, we are focusing on just two options: *smiley* and *weather* (Figure 4.2a). For this example, depending on the conditions *smiley* and *weather*, there exists four execution paths, as Figure 4.2b shows. The number in the nodes indicate the line number in the program. Figure 4.2c shows the same four executions aligned by the line numbers to clarify the differences among them. We see, for example, that in all the executions, lines 5 and 6 are always executed. In turn, lines 7 and 9 are only executed in certain configurations. Line 7 is executed when *smiley* is true, and line 9 is executed when *weather* is true. Figure 4.2d shows how the variational execution works: a single run sharing all common parts among the executions and showing the differences applied for the corresponding partial configuration space (e.g., configurations where *weather* is true).

Variational execution preserves information about the effect of inputs on control and data flow, and also shows the conditions on variables. It is able to scale to large configuration spaces due to its aggressive sharing abilities of redundancies among the executions of all configurations. As data and control flow are shared, we are able to observe feature interactions in the differences of the execution and assignments of data [50].

4.6 VAREXJ

In this work, we consider any state difference in a system that depends on more than one feature as an interaction, even if it does not cause a crash or any observable behavior differences. Such interactions describe fine grained internals of the system which may often be benign. Still, some of them can indicate faulty or unexpected behavior, such as an interaction among features that overwrite the same variables.

For the variational execution, we used and extended the variational interpreter VarexJ [50]: it is the only dynamic analysis for Java that tracks interactions on data and control flow during execution. VarexJ[3] is a variability-aware interpreter for Java bytecode based on Java Pathfinder (JPF) v7.0. JPF[4] is an extensible software analysis framework for runtime based verification purposes. In other words, it is a Java virtual machine that executes a given program not just once (like a normal VM), but theoretically in all possible ways.

Previous work on variability-aware execution [109, 110] also used JPF, but with a different proposal. They used the model checking abilities of JPF to execute separate paths (creating separate executions to each path). Conversely, VarexJ extended JPF to handle conditional values and to share as much redundant instructions as possible, thus, the software is executed just once but containing all possible execution paths. Figure 4.2d shows the conditional values for variable *c*, which may assume two different values depending on the condition *smiley.*

VarexJ provides all interactions on data and control flow for a single test case in all configurations. It also collects the presence conditions that add or change any functionality during the execution. *Presence conditions* are propositional expressions over configuration options that determine when a specific code artifact is executed [111]. We identify two main types of presence conditions: (i) *control-flow conditions*, expressions that define each path condition in a trace; and (ii) *data-flow conditions*, expressions responsible for changing the value of a given variable.

A presence condition represents a state in a variational execution. In this way, different presence conditions represent state differences between configurations. For example, *c* in Figure 4.2d can assume two values depending on the condition of *smiley.* The variable *c* can be `[:weather:]` on the condition $\neg$ *smiley* and `[:weather☺` on the condition *smiley.* A feature interaction occurs when the execution of a line of code depends on two or more features. If this execution leads to a bug, we are dealing with problematic interactions.

[3]VarexJ is available at: https://github.com/meinicke/VarexJ

[4]JPF is available at https://github.com/javapathfinder

4.7 CHAPTER SUMMARY

In our work, we address the challenge of identifying feature interactions without upfront existing specifications based on a dynamic analysis of software. Usually, real systems do not provide all specifications of the system, such as global, feature-based, and feature-interaction specifications. In addition, specifying systems requires effort and time that people are not willing to spend. We suggest an approach that aims at identifying feature interaction bugs without the need for specifications. We propose to inspect interactions based on the execution of multiple configurations of a software system (from each given test case).

Next chapter presents VarXplorer: our approach and tool to detect interactions from the variational executional of a system.

Chapter

5

VARXPLORER

In this chapter, we introduce VarXplorer, an iterative analysis that inspects feature interactions from the variational execution generated by VarexJ. Figure 5.1 shows an overview of our approach: given a configurable system and a set of test cases, we detect interactions and provide an incremental analysis of the relationship between features, illustrated through a feature-interaction graph.

To further support developers in understanding the detected interactions, we analyze control and data flow interactions to present additional indicators, such as the suppression of one feature by another. We also mark each interaction with additional helpful information, as for example the affected program variables. We present this feature-interaction graph to developers for *manual inspection*. Based on this inspection, they may indicate intended behavior and also select interactions as forbidden through the feature interaction specification language (*create spec.*). For unintended interactions, developers may go back and *fix the problem* in the code. The graph is then refined as more test cases are run, while also taking into account the documented interaction specifications (*apply on next test case*). Unlike global and feature-based specifications, interaction specifications do not specify the behavior of the system or feature. Instead, they help developers focus only on potential bugs by automatically removing benign interactions from the graph.

Thus, this chapter presents the details on how VarXplorer supports developers on the detection of suspicious interactions. The chapter consists of four main sections:

Section 5.1 presents the overview of our approach;

Section 5.2 details the detection of interactions, which has 2 main steps: pairwise detection and relationship analysis

Section 5.3 shows how we automatically create the feature-interaction language;

Section 5.4 discusses how a user interacts with the tool;

Section 5.5 presents plug-in implementation details; and

Section 5.6 concludes the chapter.

5.1 ITERATIVE ANALYSIS OF FEATURE INTERACTIONS: OVERVIEW

In Figure 5.1, we present an overview of our approach to incrementally analyze feature interactions. Given a configurable system, we execute test cases (system inputs) looking for feature interactions. The developer then explores which interactions are problematic. We support them in the process with a *feature interaction graph*, a concise representation of all (pairwise) interactions among features. Based on the variational execution of a system, the graph provides a visualization of which features interact, in addition to present their relationships and data context.

Only indicating which features interact (raw interactions) does not provide sufficient insights for the developer to identify whether a certain interaction is benign or represents a bug. For example, two features A and B may collaborate together to deliver some correct system behavior. However, under specific system inputs, the functionality provided by B may be suppressed by A in an unintended way. To understand the relationship between features, we propose to investigate the relation that a feature may have over others, such as suppressing or requiring another feature. Interaction relationships may additionally be associated with the data context of the interaction, as for example the variables involved in the relation. The different values that a given variable may assume can be a signal that something wrong occurred. Highlighting the variables involved might help developers in identify problematic interactions.

Our interaction detection process is incremental in the sense that, based on user inspection, the graph is automatically refined by removing benign interactions. This refinement is supported through a *feature interaction specification language* and ensures: (i) that the user does not see benign interactions again in future iterations (i.e., when executing other test cases); and (ii) that any newly detected unintended interactions are flagged in the future. The goal is to incrementally remove intended interactions in order to focus on unintended interactions.

Unlike global and feature-based specifications, that (respectively) represent the behavior of configurations and features, interaction specifications aim to point out that there exists an interaction between two features, without the need to formally specify its behavior. To make specifications easy to create, developers can mark interactions as either allowed or forbidden through a right click on the line that connects two features in the graph.

5.2 INTERACTION DETECTION

In the interaction detection process, we identify and analyze all pairs of features that interact in a system. The input of the detection is a variational trace created from executing a test case, and the output is the interaction graph presenting all the interactions. Our current approach focuses on pair-wise feature interactions, which has been proved an effective and practical method to test software [112, 113], specially with variational execution, where higher-order interactions are less common in practice [50].

The creation process of the interaction graph has two major steps: *pairwise detection* and *relationship analysis*. First, we identify the pairs of features that interact from the

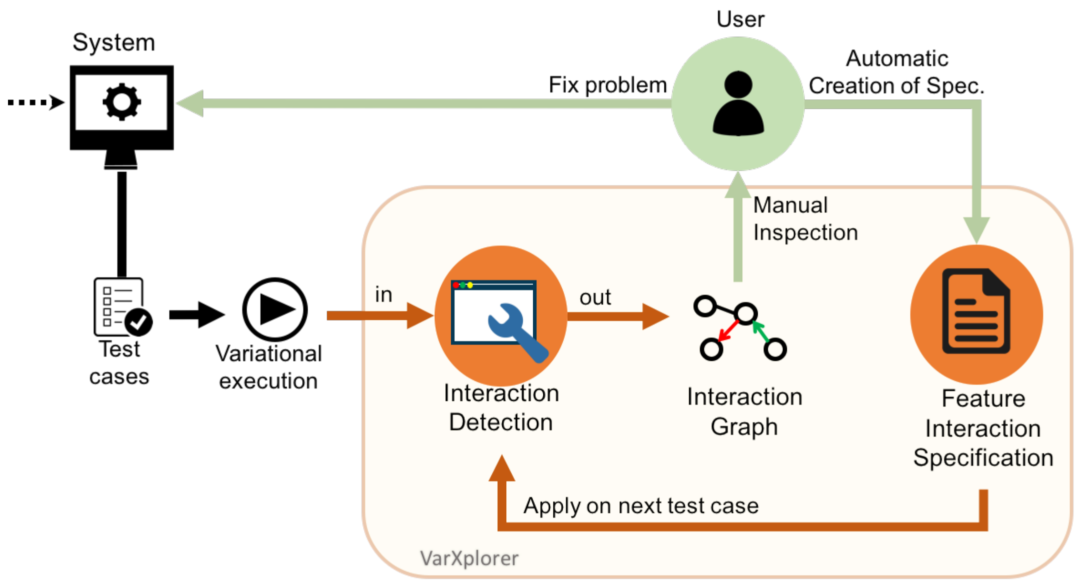


Figure 5.1: Overview of our approach to iteratively and automatically inspect feature interactions with VarXplorer

variational execution and create a basic interaction graph. Then, we perform the relationship analysis and refine the basic graph with additional information about the relationship between features, including the underlying variables they affect, to produce the complete interaction graph. This complete graph provides more details about how the features interact, whose goal is to support developers to understand problematic interactions.

5.2.1 Pairwise Detection

For pairwise detection, we collect a set $\mathbb{PC}$ with all the presence conditions in data and control flow present in the variational trace. $\mathbb{PC}$ contains all the conditions that shows how the features interact in the system. Control flow conditions are path conditions of the trace, and data flow conditions are formed by the conditions on each system variable. From $\mathbb{PC}$, we identify all pairs of features that interact together by finding feature pairs that occur together in the same condition.

Given all conditions in $\mathbb{PC}$, the set of features of a system ($\mathbb{F}$), and the set of all possible pairs of features in a system, relation $\mathbb{I}$ represents only the pairs that interact:

$$\mathbb{I} \subseteq \mathbb{F} \times \mathbb{F} \tag{5.1}$$

Given a pair of features ($f1$, $f2$), we assume that there is an interaction between $f1$ and $f2$ if there is at least one presence condition $p \in \mathbb{PC}$ in which $f1$ and $f2$ occur simultaneously as literals in p:

$$f \blacktriangleright p \quad := \quad f \text{ occurs as literal in } p \tag{5.2}$$

$$\mathbb{I} = \{(f_1, f_2) \mid p \in \mathbb{PC} \wedge (f_1 \blacktriangleright p) \wedge (f_2 \blacktriangleright p)\} \tag{5.3}$$

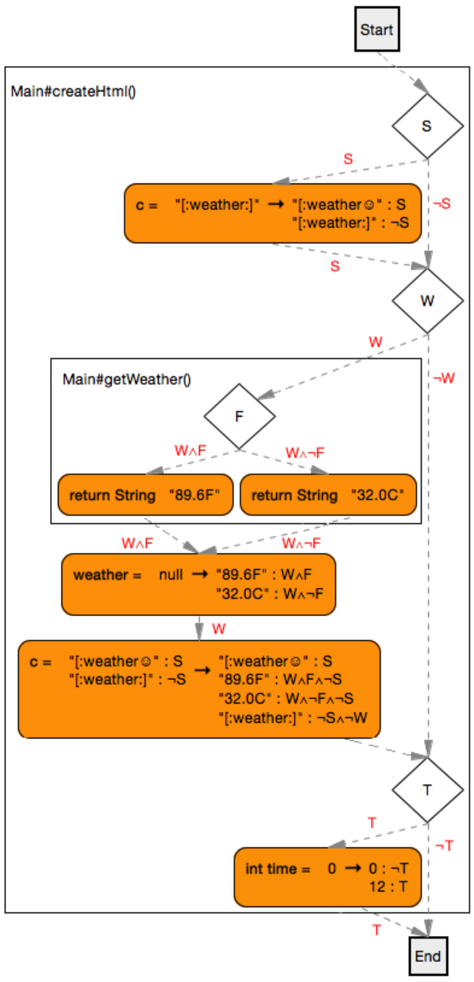

Start
Main#createHtml()
S
S
c = "[:weather:]" → "[:weather☺" : S "[:weather:]" : ¬S
¬S
S
W
W
Main#getWeather()
¬W
F
W∧F
W∧¬F
return String "89.6F"
return String "32.0C"
W∧F
W∧¬F
weather = null → "89.6F" : W∧F "32.0C" : W∧¬F
W
c = "[:weather☺" : S "[:weather:]" : ¬S → "[:weather☺" : S "89.6F" : W∧F∧¬S "32.0C" : W∧¬F∧¬S "[:weather:]" : ¬S∧¬W
T
T
¬T
int time = 0 → 0 : ¬T 12 : T
T
End

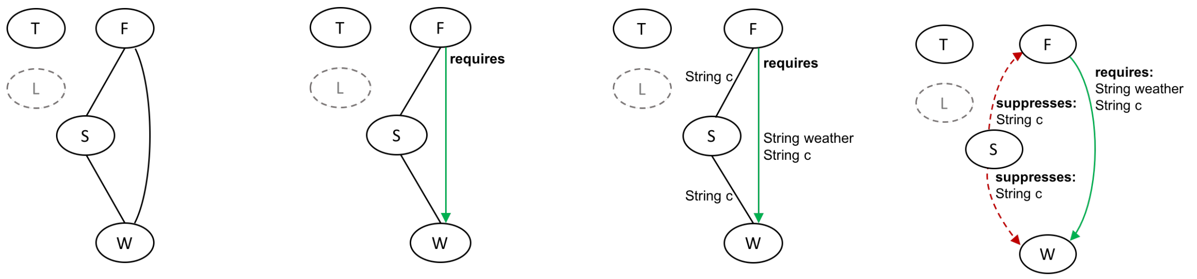
Figure 5.2: Variational trace of the WordPress example showing interactions among features. S: Smiley, W: Weather, F: Fahrenheit, T: Statistics.

From Equation 5.11 and 5.12, we are able to collect all pairwise interactions. We use them to create the *basic feature interaction graph*, a simple visualization of all interactions identified in the trace. Based on Equation 5.11, we can also determine the set of *active features* ($\mathbb{A}$) in a system, all the features that appear in presence conditions and are responsible for the functionalities executed in the system, by:

$$\mathbb{A} = \{f \mid p \in \mathbb{PC} \wedge (f \blacktriangleright p)\} \tag{5.4}$$

For example, Figure 5.2 shows the execution of the WordPress program, corresponding to the code in Listing 4.1. The figure shows the presence conditions identified during the variational execution that add or change any functionality during the execution. Control-flow conditions are showed as arrows and data-flow conditions are showed in the rounded rectangles.

The WordPress (*wp*) example has five features: `SMILEY` (S), `WEATHER` (W), `FAHRENHEIT` (F), `STATISTICS` (T), and `SECURE LOGIN` (L). $\mathbb{PC}_{wp}$ represents the set of presence conditions of the Wordpress example. For our analysis, we collected $\mathbb{PC}_{wp}$ from the variational



(a) Basic Graph (b) Flow-based analysis (c) Variables detection (d) Variables analysis

Figure 5.3: Creation process of the WordPress feature interaction graph, generated by VarXplorer. Solid black line: interaction. Dashed line: data flow interaction. Dashed line around the feature: features that has no effect in the execution. Red arrow: suppress relationship. Green arrow: require relationship.

execution showed on Figure 5.2, which contains eleven unique presence conditions, as follows:

$$\begin{aligned}\mathbb{PC}_{wp} = \{&S, \neg S, W, \neg W, T, \neg T, W \wedge F, W \wedge \neg F,\\ &W \wedge F \wedge \neg S, W \wedge \neg F \wedge \neg S, \neg S \wedge \neg W\}\end{aligned} \tag{5.5}$$

Based on the above equations, we identified four active features $\mathbb{A}_{wp}$ and three interactions $\mathbb{I}_{wp}$ in the entire set of presence conditions $\mathbb{PC}_{wp}$, as follows:

$$\begin{aligned}\mathbb{I}_{wp} &= \{(F, W), (S, F), (S, W)\}\\ \mathbb{A}_{wp} &= \{S, W, F, T\}\end{aligned}$$

Figure 5.3a shows the basic graph for our running example, illustrating the interactions in $\mathbb{I}_{wp}$. Although the program of Listing 4.1 contains five features, only three of them interact with each other, as shown by the edges in Figure 5.3a. The other two are non-interacting features; they either do not interact with any other feature during system execution (they are activated but do not interact) or are not executed in any configuration related to the current test case (they are not activated).

In our running example, the feature `SECURE LOGIN` (L) is not activated, $L \notin \mathbb{A}_{wp}$. Thus, L does not appear in any presence condition and has never been executed as part of the given test case (represented as a greyed-out dashed circle in the graph). In contrast, the feature `STATISTICS` (T) appears in the set of presence conditions, $T \in \mathbb{A}_{wp}$, but does not interact with any other feature (represented as a solid-border circle). In general, some features do not interact, because they cannot be simultaneously selected due to constraints in the variability model, or their implementations are orthogonal [50].

The basic graph (Figure 5.3a) may support developers in the detection of incorrect interactions. From the visualization, they can identify features that should not been interacting, or even missing interactions. Although the basic graph shows which features interact with each other, it does not provide enough insight on *how* features interact. We further investigate pairs of features from the graph to determine relationships that further describe the interaction. To support users in identifying problematic interactions, we also analyze the variables involved.

5.2.2 Relationships Analysis

In the relationship analysis, we investigate each pair to determine the effect one feature has on the other. In this step, we provide two complementary analysis: $\mathbb{PC}$-*based analysis* and *data-based analysis*. In the former, we explore presence conditions on control and data flow to identify which relation a feature may have over the other (i.e., either suppress or require other features). The latter is responsible for investigating variables that are controlled by more than one feature. Thus, we identify feature relationships exclusive to variables. For example, a feature *f1* may not present an overall suppression on the feature *f2*, but *f1* may suppress *f2* in relation to a given variable.

A *feature effect* specifies under which condition does a given feature have an effect on the trace. If a feature *f1* has no effect on the trace, then the selection of *f1* never adds or changes any functionality that was not present before [111]. In the basic graph of Figure 5.3a, the dashed feature L is not active and, therefore, L has no effect in the WordPress trace. Inactive features never have an effect. To the other features of the graph, T, F, S, and W (circles with solid lines), they have been executed and add functionalities to the execution. For example, to feature `STATISTICS` (S) to have an effect on the execution of Listing 4.1, the variable *time* must contain the current time and it has to be printed, i.e., lines 15-16 need to be executed.

In addition, we can analyze the effect of features on each other based on the *suppress* and *require* relationships. For example, the effect of a given feature *f1* is to have feature *f2* in the same execution. We define two effects, also called relationships, *suppress* and *require*, defined as follows:

Definition 1. *Let f_1 and f_2 be the two features of an interaction pair. We say that f_1* suppresses *f_2 when the suppressed feature f_2 has no effect if the feature f_1 is selected.*

Definition 2. *A feature f_1* requires *feature f_2 when f_1 has an effect only if the feature f_2 is selected.*

Relationship based on PC. We investigate each presence condition (on control and data flow) to detect feature effects in interactions. The feature effect is given by analyzing the effect of a given feature on the set of presence conditions. Formally, the effect of f on a condition p is given as the function $\mathbb{U}(f, p)$, as follows:

$$\mathbb{U}(f, p) = p[f/True] \oplus p[f/False] \tag{5.6}$$

The function $\mathbb{U}(f, p)$ give us the condition in which f has effect in p, using xor ($\oplus$). A feature f has no effect on p if enabling it (f as $True$) or disabling (f as $False$) it does not affect the value of p; therefore f does not have an effect on selecting the corresponding code fragment under the condition p. In other words, we say that feature f has no effect in condition p if $(f \leftarrow True)$ is equivalent to $(f \leftarrow False)$, where $(f \leftarrow y)$[1] means substituting every occurrence of f in p by y. When changing the feature to false or true, and the execution did not present any difference, it is because the feature has no effect on that program.

[1] p is implicit in this notation.

Otherwise, a feature f has an effect on p when enabling and disabling the feature in p, it presents a different result at least for one configuration, which means that different code fragments are executed. This method of verifying whether a feature is enabled or not is known as unique existential quantification [111].

For example, to determine the effect of feature `FAHRENHEIT` (F) on the presence condition $W \wedge F$, we would substitute F with *True* and *False*, as follows:

$$\begin{aligned}\mathbb{U}(F, W \wedge F) = & \\ &= p[f/True] \oplus p[f/False] \\ &= (W \wedge F)[F/True] \oplus (W \wedge F)[F/False] \\ &= (W \wedge True) \oplus (W \wedge False) \\ &= W \oplus False \\ &= W\end{aligned} \tag{5.7}$$

This confirms that on the code blocks that the condition ($W \wedge F$) hold, F has an effect iff W is selected. Similarly, we can determine the overall effect of a feature g taking in account all conditions in $\mathbb{PC}$. In this way, we need to consider the disjunction of all feature effects of g on each presence condition $p \in \mathbb{PC}$:

$$\mathbb{U}(g, \mathbb{PC}) = \bigvee_{p \in \mathbb{PC}} \mathbb{U}(g, p) \tag{5.8}$$

The result of Equation 5.8 corresponds to the condition under which a feature g has an effect on the whole system's presence conditions. For instance, we can now determine the the effect of F on the whole WordPress execution, which is given by the disjunction of all feature effects considering all the eleven presence conditions of the Wordpress (showed on Equation 5.5). Thus, the effect of F on the whole program is calculated as:

$$\begin{aligned}\mathbb{U}(F, \mathbb{PC}_{wp}) =& \{\mathbb{U}(F, S) \vee \mathbb{U}(F, \neg S) \vee \mathbb{U}(F, W) \vee \mathbb{U}(F, \neg W) \vee \\ & \mathbb{U}(F, T) \vee \mathbb{U}(F, \neg T) \vee \mathbb{U}(F, W \wedge F) \vee \\ & \mathbb{U}(F, W \wedge \neg F) \vee \mathbb{U}(F, W \wedge F \wedge \neg S) \vee \\ & \mathbb{U}(F, W \wedge \neg F \wedge \neg S) \vee \mathbb{U}(F, \neg S \wedge \neg W)\} \\ \mathbb{U}(F, \mathbb{PC}_{wp}) =& W\end{aligned}$$

In this case, it confirms that F only has an effect iff W is selected, considering the whole program, and not only one single condition. To identify explicit relationships between features (suppress and require), we can use the Equation 5.8. We say f_2 *suppresses* f_1 in an execution with presence conditions $\mathbb{PC}$ if and only if (iff) the result of the following equation is a tautology:

$$f_2 \implies \neg \mathbb{U}(f_1, \mathbb{PC}) \tag{5.9}$$

Otherwise, we say f_1 *requires* f_2 in a trace iff the result of Equation 5.10 is a tautology:

$$\neg f_2 \implies \neg \mathbb{U}(f_1, \mathbb{PC}) \tag{5.10}$$

For example, the effect of the feature FAHRENHEIT (F) on the WordPress execution results in $\mathbb{U}(F, \mathbb{PC}_{wp}) = W$. Thus, F *requires* W in order to have an effect on the system (i.e., $\neg W \implies \neg\mathbb{U}(F, \mathbb{PC}_{wp})$ is a tautology). This behavior can be observed in Listing 4.1: Line 25 is only executed when the decision in Line 8 is true, which calls the method *getWeather()* in Line 8. Then, we see that F is a sub-feature of W. From the domain knowledge, we know that this is an example of an intended cooperation in terms of a *require* relationship between those two features.

In contrast, if F would only have an effect iff $\neg W$, then W would *suppress* F (i.e., F would be blocked by W, which would be a bug). We perform the same analysis for each pair of features to determine the effects of each feature in an interaction. This analysis identify all cases of suppress and require relationships between features, which may support the user to find faulty behaviors, relationships between features that should not be allowed.

Figure 5.3b shows the result of the *relationship analysis based on* $\mathbb{PC}$ for our running example. It presents the feature effect analysis for all pairs in $\mathbb{PC}_{wp}$. In this case, we only found an explicit feature effect in the interaction (F, W), a *require* relationship. The other two interactions, (S, F) and (S, W), did not expose any explicit flow relation. Although S interacts with F and W, they do not present any flow relationship in terms of *require* or *suppress* interactions. To further explore additional relationships between features, we complement the flow analysis with a data analysis.

Relationship based on data. In a highly configurable system, the same variable can assume different values under different configurations. Features that do not directly interact on the control flow may still interact by controlling the same variables. *Conditional variables* are variables in which the values depend on more than one feature. Unexpected data values may reveal bugs from unintended interactions on variables. Conditional variables can help developers understand if a feature changes a variable value incorrectly under certain configurations, leading to a bug.

In the data analysis step, we perform two main tasks: (i) we present the data context of interactions, based on the variables they interact on; and (ii) we analyze feature effects on data to find feature relationships related to variables (e.g., a feature may suppress another related to a given variable).

Context Collection. To analyze the context of data interactions, we investigate each conditional variable and its context. A *variable context* is the set of conditions that affect the value of one variable. From the variable context analysis, we can identify all pairs of features that interact on the variable's value. To identify feature interactions in variables (data interaction), we consider the same Equation 5.12, but replace the set of presence conditions $\mathbb{PC}$ with the context of a given variable.

Let ctx_v be the context of a variable v. We assume that there is a data interaction related to v ($\mathbb{I}_{ctx-v}$) between the features *f1* and *f2* if there is at least one presence condition $pc \in ctx_v$ in which *f1* and *f2* occur simultaneously as literals in pc:

$$f \blacktriangleright pc \quad := \quad f \text{ occurs as literal in } pc \tag{5.11}$$

$$\mathbb{I}_{ctx_v} = \{(f_1, f_2) \mid p \in ctx_v \wedge (f_1 \blacktriangleright pc) \wedge (f_2 \blacktriangleright pc)\} \tag{5.12}$$

The WordPress example has three variables (`c, weather`, and `time`), but just two (`c` and `weather`) are considered as conditional variables. Since the variable `time` only depends on feature T (Listing 4.1), it is not part of any data interaction. In contrast, the context of variable `c` (ctx_c), for example, is composed of five different presence conditions, presenting combinations among F, S, and W, as follows:

$$ctx_c = \{S, \neg S, W \wedge F \wedge \neg S, W \wedge \neg F \wedge \neg S, \neg S \wedge \neg W\} \tag{5.13}$$

$$\mathbb{I}_{ctx_c} = \{(F, W), (S, F), (S, W)\} \tag{5.14}$$

The graph in Figure 5.3c shows all variables involved in WordPress' interactions, (S, F), (S, W), and (F, W). Figure 5.3c is the same graph of Figure 5.3b, but now additionally shows the variables. Equation 5.14 shows that the variable `c` is presenting in the three interactions of our graph.

Analyzing Data Relations. From Figure 5.3c, the developer is able to inspect variables that should be overwritten in an interaction, for instance. However, that graph does not provide any information on how the features behave in relation to variables. For example, we may identify cases where a feature is suppressed by another related to a given variable. To help developers understand what is happening in each variable, we detect relationships on variables and present them in the graph. Thus, we again investigate the feature effect of each feature pair, but now only related to the presence conditions of the variable being analyzed. Feature effect on data can be used to inspect each conditional variable and identify the effect it causes in the relationship between two features.

The analysis of feature effect per variable is analogous to the effect analysis for the entire set of presence conditions $\mathbb{PC}$ in Equation 5.8. The only difference is that in place of $\mathbb{PC}$, we use the context of a variable. For example, we can use the feature effect on data to investigate the variables of Figure 5.3c: interaction (F, W) related to variables c and *weather*; and interactions (S, F) and (S, W) related to *variable c*.

To investigate (S, W) according to c, we need to analyze the effect of both S and W on the context of c. Thus, we need to perform two analysis: whether the selection of S influences the value of c and whether the selection of F influences the value of c. This analysis checks if one feature *suppress* or *requires* the other feature related to variable c. For example, a given feature may only change the value of a variable when another feature is not selected (*suppress* relationship). When a feature is not selected, it has no effect in the program. Thus, in the analysis of effect on data, we also look for relationships.

In this way, to check the pair (S, W) according to c, first, we may check the effect of feature W. Given ctx_c as the set of conditions of variable c, we analyze the effect of W according to c creating a disjunction among all W effects of each condition in ctx_c (presented om Equation 5.13):

$$\begin{aligned}\mathbb{U}_c(W, ctx_c) &= \mathbb{U}_c(W, S) \vee \mathbb{U}_c(W, \neg S) \vee \mathbb{U}(W, W \wedge F \wedge \neg S) \vee \\ &\qquad \mathbb{U}_c(W, W \wedge \neg F \wedge \neg S) \vee \mathbb{U}_c(W, \neg S \wedge \neg W) \\ &= \neg S\end{aligned}$$

As a result of the disjunction $\mathbb{U}_c(W, ctx_c)$, W has effect on (or influences) variable c iff S is not selected ($\neg S$). In other words, we may say that `SMILEY` (S) *suppresses* `WEATHER`

(W) in relation to variable c. When `SMILEY` is present in the configuration, `WEATHER` effect is blocked in the program and, thus, `WEATHER` cannot override the value of variable c as it should. Second, for completeness, we check the effect of the other feature S on each presence condition of c, $\mathbb{U}_c(S, ctx_c)$, as follows:

$$\begin{aligned}\mathbb{U}_c(S, ctx_c) &= \mathbb{U}_c(S, S) \vee \mathbb{U}_c(S, \neg S) \vee \mathbb{U}(S, W \wedge F \wedge \neg S) \vee \\ &\quad \mathbb{U}_c(S, W \wedge \neg F \wedge \neg S) \vee \mathbb{U}_c(S, \neg S \wedge \neg W) \\ &= True\end{aligned}$$

The second analysis results in *true*, which means that S does not interact with another feature to affect the value of c. Similarly, we can analyze the effect of feature `FAHRENHEIT` (F) on the interaction pair (S, F) related to the variable c. `SMILEY` also *suppresses* `FAHRENHEIT` (W) in relation to c. According to the code in Listing 4.1, the variable c only gets the temperature (either 89.6°F or 32°C) when `SMILEY` is not selected. Therefore, `SMILEY` *suppresses* both `WEATHER` and `FAHRENHEIT`.

Figure 5.3d shows the complete feature interaction graph for our WordPress example, for both relationship analyses provided by our approach. Figure 5.3d is an update of the graph in Figure 5.3c, now also presenting the relationships per variable. Since those new relationships do not cover all the information of a feature, but just the variable analyzed, we call them **partial relationships** and they are represented as dashed directed arrows.

In summary, from the relationship analysis of WordPress based on both $\mathbb{PC}$ and data, we found that F *requires* W in $\mathbb{PC}$, which means that F is only executed when S is also selected. Based on the domain knowledge, that case represents a benign interaction between F and W. Besides, S *suppresses* F in data (variable c): when both F and S are selected, the variable c is not overwritten by F. This last case may be an example of a bug because wrong information is displayed to the user. Instead of seeing the current temperature, users see the tag "[:weather☺". Finally, we found that S also *suppresses* W in variable c. Then, in the presence of S, W also does not overwrite c, which presents the same wrong tag to the user.

5.3 INTERACTION SPECIFICATION LANGUAGE

The feature-interaction graph shows all the data and control flow interactions based on a variational trace. The trace shows the differences among all configurations for a given test case (specific system input). To better inspect all the possible interactions in a system, the feature interaction detection should be applied over different inputs to achieve a high system coverage. However, when applied over real systems, the graphs may present a large amount of interactions and conditional variables. In addition, different graphs from different test cases may share the same interactions. Although the input may be different, some pairs of feature may interact in the same way, as for example, overwriting the same variables with the same values.

Hence, we propose the *feature interaction specification language*. It helps developers to either allow or forbid interactions in a configurable system. When allowing, they may remove interactions from features that are intended to interact and present a benign behavior, which "cleans" the graph and can facilitate finding interactions that represent

```
<system name="WordPress">
    <specification type="allow">
        <require from="Fahrenheit" to="Weather">
            <var name="weather"/>
            <var name="c"/>
        </require>
    </specification>
</system>
```

Figure 5.4: Example of interaction specification to WordPress.

a bug. Otherwise, an interaction flagged as forbidden in a graph can be tracked throughout all test cases executions to point out the cases when it may occur.

The interaction language is a lightweight strategy to indicate that there is an interaction among features. It does not require a formal description of the behavior of systems or features, as global and feature-based specifications do. Furthermore, those behavior specifications are usually missing. Our language is then an alternative to automatically support developers in detecting bugs. For example, they can right click on the graph to specify that an interaction is intended, which is then automatically added to the specification. In particular, specifications can be created according to three parameters: type, relationship, and target, as follows:

$$\begin{aligned} \mathit{Type} &= \{\mathit{Allow}, \mathit{Forbid}\} \\ \mathit{Relationship} &= \{\mathit{Require}, \mathit{Suppress}, \mathit{Any}\} \\ \mathit{Target} &= \{\mathit{Variable}, \mathit{Method}, \mathit{Class}, \mathit{Any}\} \end{aligned}$$

The *Type* defines whether the specification either allows the interaction to occur or forbids it. The former can be used to approve benign interactions that may be repeated in most test cases of a system; and the latter can be used to flag features that should not interact. Relationship and Target correspond to refinements of specifications. The *relationship* is used to refine the specification in terms of suppress and require, and combined with a *Target*, it is possible to allow or block interactions under the scope of a method, class, or variable. Allowing any interaction between two features may be dangerous. Then, the refinements are used to specify under which conditions two features present a benign or faulty interaction.

5.4 USER INSPECTION

VarXplorer investigates interactions among features and helps users inspect unexpected interactions. From the feature interaction graph, a user can view how features interact and specify interactions. In the WordPress example, the developer can use the specification language to specify benign behaviors (*allow*). For example, Figure 5.5 shows a screenshot of the feature-interaction graph produced by VarXplorer for the WordPress example. Guided by the visualization provided by the graph, the user can automatically allow the benign data interaction between `FAHRENHEIT` and `WEATHER`, for the variables `c` and `weather`. Figure 5.4 shows the *allow* interaction specification to this example. In this

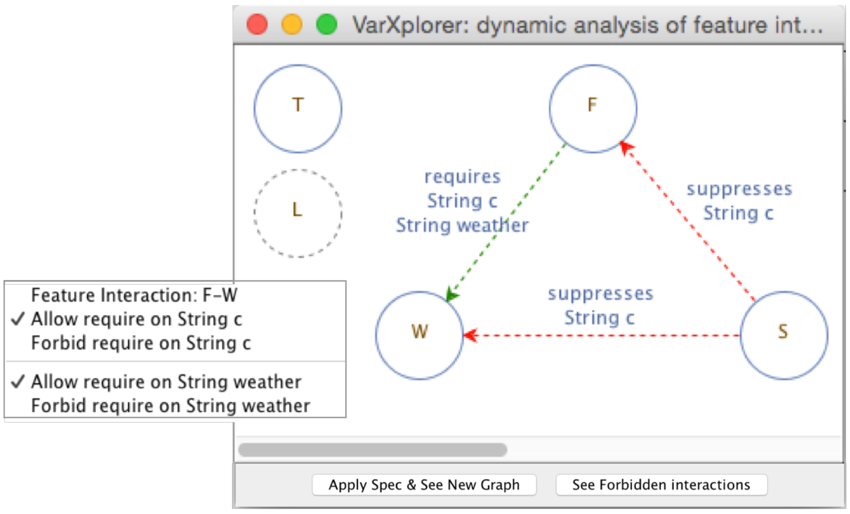

VarXplorer: dynamic analysis of feature int...
T
L
F
W
S
requires
String c
String weather
suppresses
String c
suppresses
String c
Feature Interaction: F-W
✓ Allow require on String c
Forbid require on String c
✓ Allow require on String weather
Forbid require on String weather
Apply Spec & See New Graph
See Forbidden interactions

Figure 5.5: VarXPlorer screenshot of the Wordpress graph.

way, our interaction detection approach receives the specification and guarantees that the intended interaction will not be shown again in the analysis of future test cases, which reduces the tme of analysis. The interaction between `FAHRENHEIT` and `WEATHER` is only shown again in subsequent test cases if they interact in a different way, such as on different variables or through a different relationship.

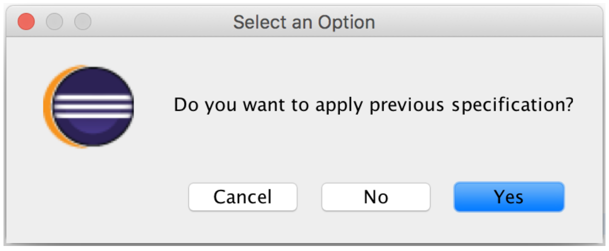
Conversely, in the other two interactions of the WordPress example (`SMILEY-WEATHER` and `SMILEY-FAHRENHEIT`), one of the features in each interaction is being suppressed by the other, which may represent a bug. In case of bugs, the user may want to fix the problem directly in the code and also mark those interactions as suspicious in the graph, by means of the *forbid* specification, as Figure 5.5 shows. Thus, if the same interactions reappear in other test cases, our tool will point them out as potential problematic interactions.

Our tool was developed to work incrementally in case of developers have a set of different test cases to execute. VarXplorer runs each test case at a time and before showing the corresponding interaction graph, it asks developers whether they want to apply previous specifications on the new execution, as Figure 5.6 shows. Hence, the tool is able to read previous specifications and apply the changes: either removing benign interactions or highlighting forbidden interactions, if any.

To support developers when using the VarXplorer plug-in, it also has two additional buttons, as Figure 5.5 shows. The first one, "Apply Spec & See New Graph", allows users to apply the options selected in the graph related to either *allow* or *forbid* interactions (interaction specifications). When pressed, it shows how the graph meets the specifications applied. The second button, "See Forbidden interactions", shows the list of previously forbidden interactions to that system.

5.5 PLUG-IN IMPLEMENTATION

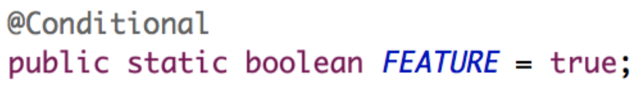
VarXplorer analyzes highly configurable programs developed in Java. Therefore, our tool needs to know which are the features of the program. Thus, features are annotated as



Do you want to apply previous specification?

Figure 5.6: VarXplorer window to confirm the use of previous specifications.

conditional values in the program. In this case, features are represented as static Boolean variables (two states, *true* and *false*). To be understood by the variability-aware interpreter, each feature variable should be initialized as conditional using the annotation `@Conditional`. The *jpf-annotations* library is used to annotate the JAVA code. Figure 5.7 shows an example of a feature variable. Although a feature is always initialized as *true*, the interpreter replaces the initial value by a choice: whenever the field `FEATURE` is used it can assume both values, *true* and *false*, depending on the current context of the execution.



```
@Conditional
public static boolean FEATURE = true;
```

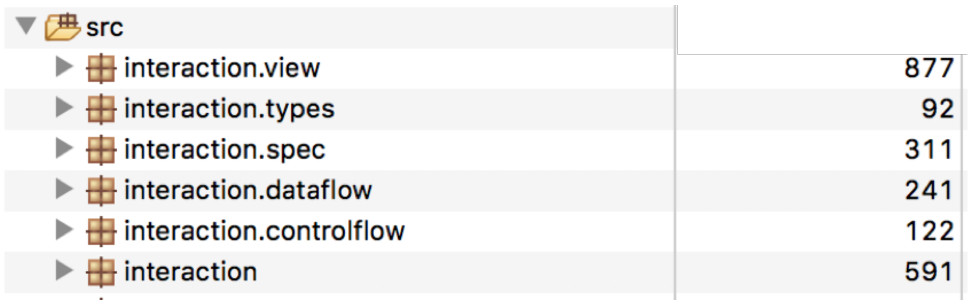
Figure 5.7: Annotated feature variable in JAVA.

VarXplorer is an Eclipse plug-in to detect interactions and graphically show them. It produces a graph that represents interactions and relationships between features. VarXplorer analyzes control and data flow information from the variational execution provided by VarexJ. The tool is publicly available online at GitHub[2]. The repository contains all sources required to execute the tool.

VarXplorer extends VarexJ in 6 packages, 33 classes, and 2234 LOC in total. The packages are shown on Figure 5.8 and described as follows:

- **interaction**: the core of VarXplorer and the interface between VarexJ and the new features provided by our tool. This package contains 8 classes.
- **interaction.view**: the user interface of VarXplorer. It contains 11 classes.
- **interaction.types**: it groups 3 classes that defines types of interaction, i.e., data flow interactions and control flow interactions.
- **interaction.dataflow**: it controls data flow interactions. This package is composed of 4 classes.

[2]https://github.com/larirsoares/VarXplorer



src	
interaction.view	877
interaction.types	92
interaction.spec	311
interaction.dataflow	241
interaction.controlflow	122
interaction	591

Figure 5.8: VarXplorer packages and LOC

- **interaction.controlflow**: it controls control flow interactions and has 1 class.
- **interaction.spec**: it contains 6 classes responsible for creating and managing specifications.

For the visualization, VarXplorer is currently using the JGraphX library[3]. It provides functionality for visualization and user interaction with node-edge graphs (not charts). Furthermore, to deal with specifications, we transform user mouse events on a XML file, as presented on Figure 5.4.

For reasoning about features, VarXplorer uses a feature expression library, named FeatureExprLib[4] of the TypeChef [114]. The library allows to easily express and reason about expressions in propositional logic, integrated with a SAT solver and Binary Decision Diagram (BDD). [5] It supports also parsing feature expressions and loading entire feature models (in textual format or a .dimacs file). For reasoning, internally both BDDs and SAT solvers are used which allows to scale reasoning even to feature models the size of the Linux kernel [114]. VarXplorer use FeatureExprLib with BDDs for handling features.

VarXplorer collects all presence conditions generated during the variational execution, and identifies which features appear together. For each pair of features, it analyzes *suppress* and *require* relationships. The Algorithm on Listing 5.1 shows how we detect pairs and relationships. First, we use BDD to get the set of valid features (Line 4). From the set of expressions (gathered from VarexJ), we identify each pair of feature that interact (line 6). We also identify features that have no effect in the trace of execution (line 8). Then, we combine the features in pairs to look for relationships (lines 10 and 18). This investigation was explained on Equations 5.6 and 5.8, in which we analyze the effect of each feature present in a given program (line 12). Finally, we check the implications to detect relationships on a pair of features, described on Equation 5.9 for suppress and Equation 5.10 for require, respectively shown on lines 24 and 25 in Listing 5.1. This algorithm returns all pairs of features that interact and relationships, if any.

[3]JGraphX is a Java Swing diagramming (graph visualization) library, available at https://github.com/jgraph/jgraphx

[4]FeatureExprLib is available at https://github.com/ckaestne/TypeChef/tree/master/FeatureExprLib

[5]BDD is a data structure that is used to represent a Boolean function. On a more abstract level, BDDs can be considered as a compressed representation of sets or relations.

The method `getExpressionsPairs` presents the highest algorithmic complexity (line 6) with an approximated value of $\texttt{O(e.n}^2\texttt{)}$. The literal `e` represents the number of expressions and `n` represents the number of features. If the number of expressions is close or higher than the number of features, the complexity of the method tends to be cubic $\texttt{O(n}^3\texttt{)}$ or even higher. Conversely, if the number of expressions is much lower than features, the complexity presents a quadratic behavior ($\texttt{O(n}^2\texttt{)}$). Since the method `getExpressionsPairs` presents the highest complexity of the algorithm, the final complexity is determined by this method.

The detection of relationships on variables (partial relationships) is similar to the algorithm on Listing 5.1. The main difference is the set of expressions received by the algorithm. For partial relationships, we run that algorithm for each variable, passing to it the subset of expressions of the given variable. To find all the conditional variables, the ones that depend on more than one feature, we extended VarexJ to collect variables and their expressions during the variational execution.

5.6 CHAPTER SUMMARY

In highly configurable systems, features may interact unexpectedly, producing faulty behavior. We propose VarXplorer, an incremental and iterative lightweight process and plugin to detect problematic interactions dynamically. From a variational execution, we gather all the variability context (control-flow paths and shared data) in which each instruction is executed to create a feature interaction graph. VarXplorer uses this information as input to identify how the features are related to each other and helps users to inspect unintended interactions. While analyzing the graph, users may indicate interactions that present a benign behavior and also mark others as forbidden, through the feature interaction specification language.

Next chapter presents detail about our first study, a controlled experiment to evaluate our approach.

Listing 5.1: Simplified algorithm for the detection of interactions and relationships

```

getInteractions(expressions){

    featuresSet = BDD.getFeatures;//get the whole set of features

    exprPairs = getExpressionsPairs(expressions);//get all the pairs that appear
        together in expressions

    noEffectlist = getNoEffectlist(features, expressions);//list of features that
        do not appear in the expressions

    for each feature1 in featuresSet {//get the 1st feature for pair comparison

        unique = createUnique(feature1, expressions);//get the effect of a given
            feature1 on the set of expressions

        if (isContradiction(unique)) {//when a feature doesn't appear in the
            expressions
            continue;
        }

        for each feature2 in featuresSet {//get the 2nd feature for pair comparison

            if (isTautology(feature2) {//when the feature is the feature model root
                feature
                continue;
            }

            suppressAnalysis = feature2.implies(unique.not());//checking the
                implication for suppresion
            requireAnalysis = feature2.not().implies(unique.not());//checking the
                implication for requirement

            pair = new PairExp(feature1, feature2);//creating the pair

            if (suppressAnalysis.isTautology()) {
                phrase = feature2 suppresses feature1;
            }
            if (requireAnalysis.isTautology()) {
                phrase = feature1 requires feature2;
            }

        }

        if relationships were not found for a pair{
            phrase = feature1 and feature2 do not interact;
        }

        interactionList.add(pair, phrase);//adding pair and relationship in a list
    }
    return interactionList;//final list of relationships
}
```

PART IV

EMPIRICAL STUDIES

Chapter

6

CONTROLLED EXPERIMENT: UNDERSTANDING FEATURE INTERACTIONS WITH THE GRAPH

In order to understand how feature-interaction graphs provided by VarXplorer can help users identify suspicious interactions, we performed a controlled experiment. This study investigated and compared the ability of users to identify problematic interactions with and without VarXplorer, in a setting with different tasks and systems.

The experiment was executed with 24 participants from different universities and companies. We measured the effort to identify a buggy interaction based on the information provided by the feature-interaction graph. We also performed a qualitative analysis based on video and audio recordings, and post-treatment interviews. The results showed that participants using VarXplorer outperformed participants using the state of the art tool. From the qualitative analysis, we also identified and discussed 5 observations, including how the feature relationships support identifying bugs.

This chapter consists of three major sections:

Section 6.1 presents the experimental design of our empirical study, such as research questions, pilot study, participants, tasks, and procedure;

Section 6.2 discusses and analyzes the main findings of our empirical study,

Section 6.3 presents the threats to validity of our study;

Section 6.4 presents the related work; and

Section 6.5 concludes the chapter.

6.1 EXPERIMENTAL DESIGN

This section shows how we conducted the experiment. It presents details about research questions, participants, pilot study, tasks, design and execution, as follows.

6.1.1 Research Questions (RQs)

We investigate the usefulness of interaction graphs as a strategy to identify feature interaction bugs in programs. VarXplorer is an Eclipse plug-in that abstracts from multiple executions to show feature relationships. To the best of our knowledge, there is currently no comparable tool that is able to detect suspicious interactions based on a dynamic analysis of relationships between features, without specifications.

One possible baseline could be to compare VarXplorer with a traditional source-code inspection or the standard Eclipse debugger. However, we assume that inspecting the code to detect interactions is hard, slow, and possibly a tedious work. On the other hand, the standard debugger is a general-purpose tool that is not specifically designed with feature interactions in mind.

Recent work already shows that Varviz, an Eclipse plug-in that provides a variability aware execution trace of the code, outperforms the standard debugger for comprehension tasks involving interactions [65]. Based on such previous findings, we compare VarXplorer to the current state-of-the-art tool in this area, VarViz. Like VarXplorer, Varviz is an Eclipse plug-in. Varviz enables programmers to use variational traces for debugging interaction faults. It provides a trace of execution, instead of code or debug. The Varviz trace is present in Figure 5.2.

We aim to answer the following main question: *Does VarXplorer help developers identify suspicious feature interactions?*, which we split into two concrete research questions:

- RQ1: Does VarXplorer improve the performance of identifying suspicious interactions compared to Varviz?
- RQ2: How does the interaction graph presented by VarXplorer help understand the suspicious interactions in a program?

RQ1 is related to the effort required to identify suspicious interactions. We measured the time spent to detect interactions in two setups: using VarXplorer and Varviz. For each tool, we created two tasks for two different systems. We measured how long participants take to identify and understand a suspicious/buggy interaction from the information provided by (i) the graph generated with VarXplorer; versus (ii) the execution trace created by Varviz.

To answer RQ2, we analyze what information helps participants understand and identify suspicious interactions. In addition, we want to know how the graph components (relationships, variable arrows, and colors) can help developers with the detection of buggy interactions.

6.1.2 Experiment Overview

We designed our experiment as a within-subjects study. For this design, the same group of participants receives more than one treatment [115]. In this way, all participants perform tasks using both tools, VarXplorer and Varviz. The tools are the treatments of our experiment.

Within-subjects designs have greater statistical power than between-subjects designs: we need fewer participants in the study to find statistically significant effects, because each participant is tested under all treatments. Within-subjects designs also represent a good strategy when it is difficult to recruit participants [116].

The experiment consists of two tasks: participants first start with one tool and they have to identify interactions in a given system. After finishing the first task, they start the activities with the second tool and another system. For each tool, they use a different task to reduce learning effects.

While the participants are working on the tasks, we ask them to verbalize their thoughts and tell us what they are doing (think-aloud protocol [117]). When necessary, we also ask them why they are doing a particular activity. The think-aloud protocol makes the process as explicit as possible during the tasks, because it captures preference and performance data simultaneously, rather than waiting until the experiments finishes to ask all the questions. In addition, we record the screen and audio to collect supporting data for analyzing the time and strategy used by participants to find interactions. We run the experiment for one participant at a time.

We complement the above setup with a pre-questionnaire and a post-interview. Before the experiment, we ask them to answer an online pre-survey[1], which we used to collect background data about their experience, mainly with Java and the Eclipse IDE. We create balanced groups of participants based on their experience. In this way, related to Java development, we have a mix of experts, intermediate, and beginners participants in each group. For the post-interview, we asked two questions: (1) which tool is easier to understand an interaction? and (2) what makes this tool easier in comparison to the other one? We triangulate the gathered answers with the data we obtain from the think-aloud protocol.

6.1.3 Pilot Study

Before the main experiment, we conducted two pilot studies with 8 graduate students from two universities in different countries, Brazil and the US. We used the pilot study results to determine the amount of time needed to execute our tasks. This allowed us to estimate and plan the number of participants we needed in the main study. We found a large effect size in the difference between the participants who used VarXplorer (3 min on average) versus the ones who used Varviz (13 min on average), which suggests that we do not need a large group of participants. The pilot study also allowed us to assess whether the participants could properly understand the subject systems and the tasks they should perform, as well as to train the researcher who overlooked the experiment. We do not consider the results of the pilot in our analysis.

6.1.4 Participants

After the pilot, we recruited 24 participants (excluding pilots). To recruit them, we sent emails to professors in two universities, from different computer fields, to suggest former

[1]It is a Google Form document. We have a copy in the Appendix B.1.

Table 6.1: Participants

Partic.	Institution	Position	Group	Exp. (years)
1	U1	M	1	>= 5 and <10
2	U1	PhD	1	>= 10
3	U2	Un & dev	1	>= 1 and <5
4	U1	M	1	<1
5	U1	PhD	1	>= 1 and <5
6	U2	Un	1	>= 1 and <5
7	U1	PhD	2	>= 10
8	U1	M & dev	2	>= 5 and <10
9	U1	Un	2	>= 1 and <5
10	C1	Dev	2	>= 1 and <5
11	U1	M	2	>= 1 and <5
12	U1	M	2	>= 1 and <5
13	U1	M	3	>= 5 and <10
14	C2	Dev	3	>= 10
15	U1	M	3	>= 5 and <10
16	C3	Dev	3	>= 1 and <5
17	U1	Un	3	>= 1 and <5
18	U1	Un	3	>= 1 and <5
19	U1	M	4	>= 5 and <10
20	U1	M	4	>= 1 and <5
21	U2	Un & dev	4	>= 1 and <5
22	C4	Dev	4	>= 5 and <10
23	U1	Un	4	>= 1 and <5
24	U1	PhD	4	>= 10

students (developers) and current ones.

We received 24 emails from three different profiles: undergrad students, graduate students, and professional developers. Furthermore, some of the students are also developers. The participants experience regarding Java and Eclipse IDE varied from few months to more than 10 years.

Table 6.1 shows the participants involved in the experiment: 7 undergrad students (Un), 9 master students (M), 4 PhD. students (PhD), and 7 developers (Dev). The students are from two different universities (U1 and U2) and the developers work in four different companies (C1, C2, C3, and C4). According to our design, we created four groups with a similar background distribution of participants.

6.1.5 Experimental Material and Tasks

We used two product lines as the evaluation material: Elevator and Telephone.

The elevator system has been proposed by Plath and Ryan [88]. It is an extensible elevator model whose features are designed to highly interact. For example, the elevator needs to stop if it is empty or priority service for a special floor is activated. Although this system has only 1046 LOC and 6 features, it is hard to understand the impact of its features due to the interactions. Furthermore, it has been frequently used in the literature [118, 119, 120]. We used the Elevator Java version from the SPL2go repository[2].

The telephone system has been widely discussed in the literature due to the Feature Interaction Detection Contest that was held in 1998 and 2000 [121]. The contest aimed to compare various methods and tools for detecting feature interactions. To enable a

[2]http://spl2go.cs.ovgu.de

comparison, the objective was to detect interactions among a given set of features for a given telephone system. The telephone system was designed to present many interactions. Based on the specification from the contest, we created a Java implementation for the telephone system. We implemented 6 features and 1005 LOC.

We designed two tasks, one for each system. The tasks were designed to be similar in size, number of features, and time to be executed. The pilots served to align them. In general, we asked the participants to use the tool given to them (either VarXplorer or Varviz) to identify suspicious interactions on the systems for a given test case. The tasks were designed to present just one suspicious interaction for each system and a couple of benign interactions. We provided the participants with the description of each feature in the target system, test case scenario documentation, and the system's source code[3]. From those artifacts, they get the domain knowledge about the systems. Thus, the participant role in the experiment is to identify the problematic interaction in each system. We next describe the details of the two tasks.

Task 1. According to the features specification, when the elevator has two thirds of the maximum weight, it should not attend to calls until it delivers passengers, making the weight be less than two thirds. However, because of a problematic interaction between two features (*Executive Floor* and *Two Thirds Full*), the elevator goes to pick a passenger up even though it has already achieved two thirds of the capacity, which forces the elevator to not close the door until someone leaves it. In this situation, the feature *Executive Floor* is blocking the execution of the feature *Two Thirds Full.* In this task, the participants should figure out that this interaction leads the system to a wrong behavior. They have to identify the suspicious interaction using either Varviz or VarXplorer, depending on the group they were assigned. We request them to identify the problem, but we do not require them to fix the problem in the source code.

Task 2. The contest instructions describe all the feature specifications [103], such as: (i) *Call Forward on Busy*, all calls to the subscribing line are redirected to a predetermined number when the line is busy; and (ii) *Call Waiting*, allows the subscriber to be noticed that another party (incoming call) is trying to reach his number while her line is busy. However, when the line is already busy and another number is trying to reach that line, because of a precedence in the implementation, the telephone system is always forwarding the call, even if the person wants to put the incoming call on waiting. Again, one feature has its behavior suppressed by another and the participants should identify that this is the suspicious interaction, by using one of the two tools.

6.1.6 Design

The tool used during the experiment represents an independent variable with two levels, VarXplorer and Varviz. We also distinguish participants related to the systems they use: Elevator system and Telephone system. Furthermore, to analyze the performance of the tools (RQ1), we measured the time spent by participants to find the suspicious interaction in each system. The time was measured based on the video recorded during the experiment. Time is a dependent variable of our evaluation.

[3]The documentation and tasks description is in the Appendix B.2.

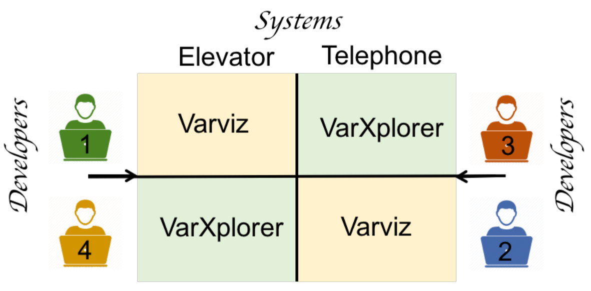


Figure 6.1: Latin Square to our treatments

Latin Square Design. Since participants perform two tasks, one followed by the other, there can be problems of carryover effects. Thus, each measurement may depend not only upon the treatment given but also on the preceding treatment [116]. To avoid those kind of effects, we use a Latin Square design [122]. It represents a method of placing treatments so that they appear in a balanced fashion within a square block. Latin Square is an useful design where the experimenter desires to control variation in two different directions. In this way, treatments should appear once in each row and column.

In addition to the standard Latin Square, we used three strategies to avoid learning effects: (i) we have every treatment preceding every other treatment the same number of times (counterbalanced Latin Squares); (ii) we change the order participants use the tools; and (iii) participants do not repeat the same tool or the same system in different tasks.

Figure 6.1 was inspired by the Latin Square to show the distribution of the population to our experiment. The columns are labelled with the two subject systems (Elevator and Telephone). The rows correspond to the developers. The 4 squares (cells) contain the two treatments (Varviz and VarXplorer). Then, we allocated one group of participants to each cell. Based on this design, each participant received the two treatments listed in a given row for the two subject system listed in the corresponding columns.

Clearly, we can only have a participant looking for interactions in a given program once, otherwise there would be a learning effect on subsequent attempts. Following the strategies of our design, participants are using different tools and systems for each task and they have never used neither the tools nor the systems before the experiment. Moreover, since we permute the order in which they perform the activities, we create 4 groups, as Figure 6.1 shows. We balanced the groups based on the participants experience. The order of each group is described as follows:

- Group 1: first Varviz-Elevator, then VarXplorer-Telephone
- Group 2: first Varviz-Telephone, then VarXplorer-Elevator
- Group 3: first VarXplorer-Telephone, then Varviz-Elevator
- Group 4: first VarXplorer-Elevator, then Varviz-Telephone

6.1.7 Procedure and Execution

Before the participants receive their tasks, we first introduced the experiment with a *tutorial* about feature interactions. The tutorial took 10 minutes on average. Then, each participant had two tasks to accomplish, with descriptions and instructions provided for each task.

Before each task, we conducted a *warm-up section* to introduce the tool (either Varviz or VarXplorer) using a third system, the mock WordPress shown on Listing 4.1. For the first warm-up, we give them Eclipse with the first tool (depending on the participant group), the mock WordPress source code, and a list of features. At this point, they have to identify the suspicious interaction in WordPress using the tool given to them. During this warm-up, we answer their questions about the tool. After that, we give them the *first experiment task* corresponding to that tool, which includes again the Eclipse with the tool, the experiment system source code (either Elevator or Telephone) and the list of features. Those steps correspond to the first part of the experiment.

The second part starts when the participants finish the first task. Hence, we perform a second warm-up with the second tool and WordPress, where they again have to identify a suspicious interaction. After they finish the warm-up with the second tool, they start to perform the *second experiment task*, which consists of identifying the suspicious interaction in the second tool and the second system (either Elevator or Telephone). The experiment finishes when they have performed both tasks, i.e., when they finish the judgment about the interactions: whether they are benign or problematic. As final step, we conduct the post-experiment interview. Figure 6.2 shows the experiment setup for each group, and Table 6.1 shows the distribution of participants in the groups.

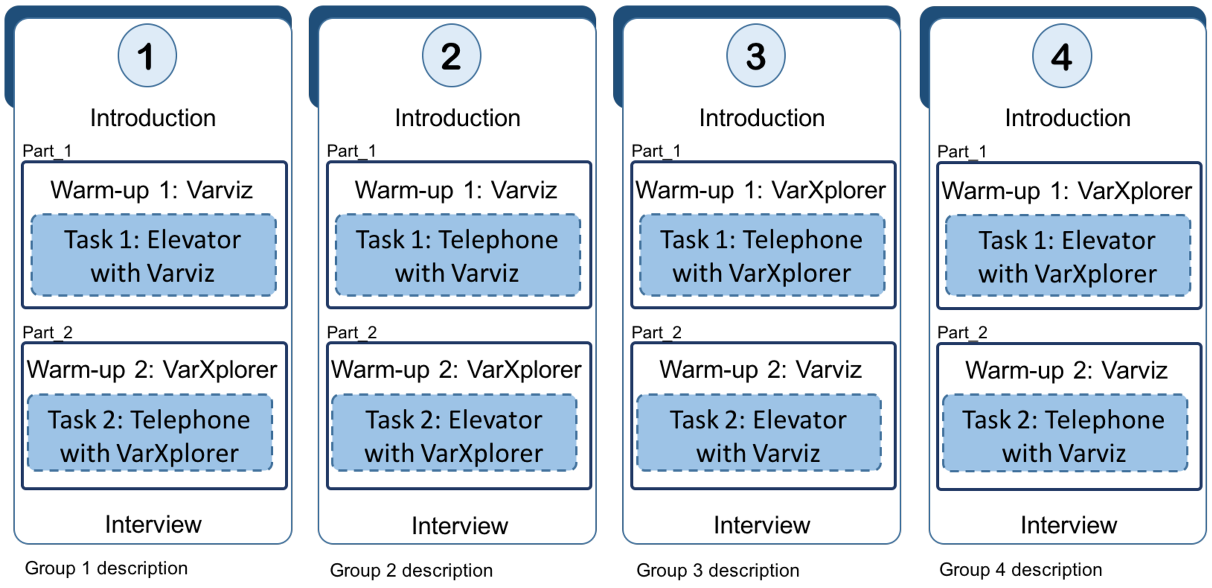


Figure 6.2: Description of the setup for each group.

To provide the same environment to each participant and avoid having to install and

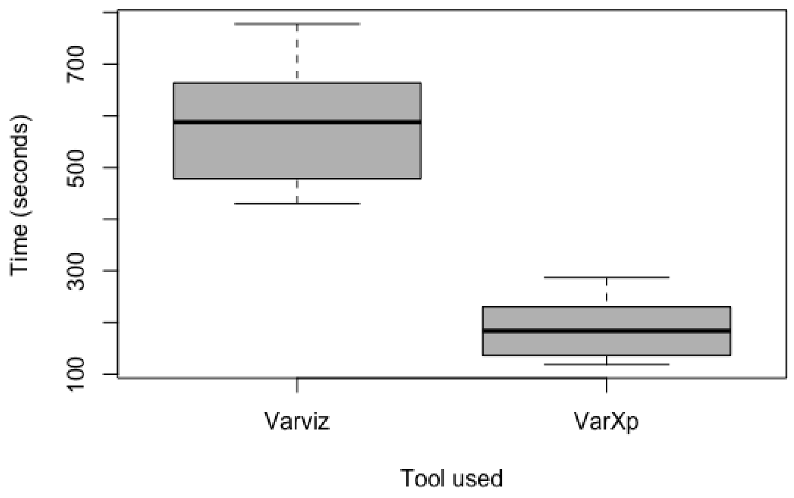


Figure 6.3: Time results for the tools

configure Java and Eclipse parameters, we used the same laptop for all participants in the experiment. Each group received two versions of Eclipse, one version with the Varviz installed and the other one with VarXplorer, with two programs in the workspace, the warm-up program (WordPress) and the experiment program (either Elevator or Telephone). Thus, each Eclipse installation corresponds to a cell of our Latin Square design, and we could make sure they would use the right tool in the right system.

6.1.8 Data Analysis

For the statistic analysis of our data, we conducted an analysis of variance using a within-subjects ANOVA. It is a parametric test for determining whether significant differences occur in an experiment containing two or more conditions. ANOVA has three assumptions: the dependent variable measures normally distributed interval, the population has homogeneous variance, and each cell (Latin Square cell, in our case) contains an independent sample [115]. We used the Shapiro-Wilk normality test, the Bartlett test of homogeneity of variances, and the Tukey HSD test to the multiple comparisons of means. We conventionally reject our hypothesis when *p-value* < 0.05.

For the qualitative analysis, we watched the videos and listened to the audios (including experiment and post-interview) looking for commonalities and differences in the way participants execute the tasks and their perception about the tools. We transcribed participants answers and informally generated codes to passages of the data which are relevant to understand participants difficulties and meaningful differences about the two tools used.

6.2 RESULTS AND DISCUSSION

This section presents the results of our experiment and discusses the implications. We next present both statistical and qualitative analysis to answer our research questions.

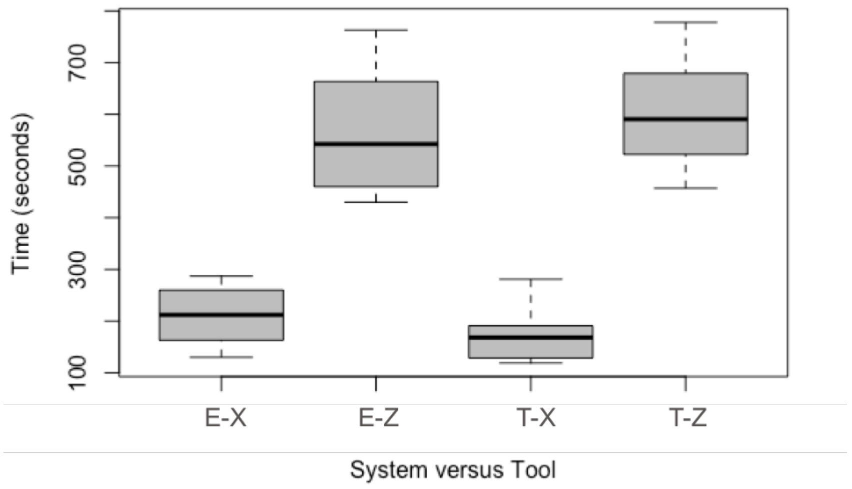


Figure 6.4: Time results grouping tools and systems. E: Elevator; X: VarXplorer; Z: Varviz; T: Telephone

6.2.1 RQ1: Does VarXplorer improve the performance of identifying suspicious interactions compared to Varviz?

Participants using VarXplorer outperformed participants using Varviz with respect to the average task time[4]. On average, participants accomplished their tasks 3.06 times faster using VarXplorer. The participants that used VarXplorer took an average of 3 min to perform the task, while the others that used Varviz had an average of 9 minutes. All the participants were able to identify the suspicious interaction in both tasks, which is why we compare time and not also correctness. Thus, all of them finished successfully their tasks. Figure 6.3 graphically shows the time results.

Statistic analysis of performance. We used ANOVA to statistically evaluate the tools. The difference between the average times to perform the study tasks with each tool proved to be statistically significant. Based on the ANOVA test, we rejected the null hypothesis (p-value $< 2e\text{-}16$) that the distribution of the population is homogeneous. Thus, VarXplorer reduces developer effort to identify suspicious interactions in both tasks, elevator and telephone system.

Figure 6.4 shows the results for our 4 groups. For both systems, there is a significant effect size between Varviz and VarXplorer tasks. The subject systems had similar performance times, and regardless of system, VarXplorer was faster.

Test of assumptions. The ANOVA test requires two assumptions of the underlying data: normal distribution and homogeneous data. Our statistical tests show that our data is normally distributed (p-value = 0.4447), but is not homogenous (p-value = 0.0015). However, the heterogeneity of the data does not affect the results of ANOVA, since the groups have the same size [115], 24 measures each[5]. Figures 6.3 and 6.4 also show that

[4]The time measured for the participants is in the Appendix B.3 and the R script is in the Appendix B.4.

[5]To confirm our analysis, we also performed non-parametric tests with Kruskal and Wilcox. On both tests, we rejected the null hypothesis, affirming that our results are robust.

the data presents a large effect size, such that violating the assumption is unlikely going to change the decision of rejecting the null hypothesis.

Analysis of order influence. Even though we designed our experiment to avoid learning effects and tool/system order influence, we still performed the ANOVA test on the groups to check whether the order presented an influence.

For the systems group, the data from the order of the systems are not different, i.e., the order of the systems does not statistically influence the results (p-value = 0.803). For the tools groups, it presents a large effect size between the groups that used VarXplorer against the Varviz groups. According to the ANOVA test, we get statistically significant evidence that our groups have different averages (p-value < 2e-16). Thus, the order of the systems does not matter to the evaluation.

In order to analyze the interactions in the tools order group, we performed a Tukey HSD test [115]. We saw a small learning effect when Varviz is used after Varxplorer (p-value = 0.0378). This situation occurs because the participants learn from VarXplorer graphs: they learn about relationships between features and start to explicitly look for them in the Varviz trace. Although the systems presented a small difference, this situation does not significantly affect the analysis of variance: this effect is tiny compared to the overall effect size. The fastest Varviz time is still significantly slower than the slowest VarXplorer time.

RQ1: The results confirmed that participants using VarXplorer identify feature interactions at least 3 times faster compared to participants using Varviz.

6.2.2 RQ2: How does the interaction graph presented by VarXplorer help understand the suspicious interactions in a program?

We analyzed the videos (audio and the screen recordings) of all participants to know how feature-interaction graphs help understand feature interactions. We watched the videos to find common activities that the participants performed during the tasks, besides comparing the findings with the interviews answers. Thus, we could compare the participants perception with the activities performed in the tasks.

Observation 1: *The explicit type of relationship for a pair of features guides the analysis and decreases the analysis time.*

VarXplorer represents an alternative to detect interactions with no need to debug the code. The *require* and *suppress* relationships graphically represented as colored arrows in the graph caught the attention of the participants to what is happening with a given pair of features. In a debugging tool, such as Varviz, subjects need to follow the execution flow step by step to interpret what is going on in the system based on methods and variables calls, for example. During the survey performed after the experiment, P20 (see Table 6.1) affirmed: *"VarXplorer is simpler and easier because it shows, in addition to the interactions, the relationships"*. Along the same lines, P13 said: *"The colors of the arrows in VarXplorer serve as an alert to me to investigate whether the interactions are correct or not"*.

Participants also talked about partial relationships. For example, when a feature suppresses the other of changing a specific variable. Partial relationships affect just one or some variables (also called *conditional variables* as discussed on Section 5.2.2), but not all the behavior of a given feature. A program usually has many variables, which may assume many values to different interactions. Looking for conditional variables is a hard task. VarXplorer makes this process faster to show the influence of a relationship on variables, interaction-dependent variables. For instance, P14 stated: *"I don't need to look for the variables which may be problematic, the VarXplorer graph already brings this information to me."*

Observation 2: *To use VarXplorer, the user might not need to know details of the implementation, or even the programming language used.*

The VarXplorer graph only presents interaction information, without showing other unrelated details, such as control flow paths, methods and classes names, non-related variables, and variables values. Any person that has knowledge about the system requirements and the feature specifications may be able to judge whether the features behave as expected, based on the relationships presented in the graph. For example, P21 said: *"VarXplorer is more objective in showing the interactions. I do not need to worry about low level of the system, such as methods, classes, components, and all the possible ways the program can go to realize that an interaction is suspicious."*

We observed that participants become convinced an interaction is suspicious based on the perception they have of the features description of the system. Figure 6.5 shows that just two of the participants looked at the source code. They looked at it during the Varviz tasks to see how some features were implemented. In this way, VarXplorer can be used by different profiles in a development team (e.g., engineers, managers, testers, and developers), even those that do not know details about the system. P8, who is a developer affirmed: *"I even do not need to know the programming language or how the code is implemented, any person from our team can use the graph to understand what is going on in the system"*.

Observation 3: *VarXplorer also shows non-interacting features and no-effect features, which might be indicatives of bugs.*

Besides interactions, VarXplorer also shows features that do not interact with any other feature (non-interacting features). In cases where developers know that a given feature should interact, this information can alert them that something may have happened, leading the feature to not interact with anyone else.

The same occurs with features that are not called in the execution (no-effect features): if we run a test case that a given feature should be executed, and it is not, it is a case to be investigated. This information about non-interacting features and no-effect features is not explicit in Varviz, while VarXplorer graphically represents them. Non-interacting features are shown in the trace because they have effect and are part of the execution, but no-effect features are not represented. This information was perceived as useful by some participants, as P16 stated: *"VarXplorer shows me features that have no effect or do not interact, while this information is hidden in Varviz"*.

Observation 4: *Understanding interactions in Varviz takes longer than finding out where* *they are located in the trace.*

Varviz tests all the combinations of features and captures control and data flow of a given test case. Since the trace shows more information than the graph, we suspected that the time to *identify* a suspicious interaction (find where in the trace the features are interacting together) would be longer than to properly *understand* the interaction (tell us if the interaction is either OK or suspicious).

Furthermore, participants complained about the time spent to find an interaction in the Varviz trace. For example, P5 said: *"I see how the trace works, but it is not clear to me where I should take a look at to find interactions"*. Another one, P7, also stated: *"Varviz presents everything together, a mix of information. It shows more things than I need to understand the interaction."*

However, we observed that most of the time was spent on properly understanding the interactions, which in practice consists in comprehending Boolean logic expressions related to the suspicious interactions present in the trace. Figure 6.5 shows the execution of the tasks for the 24 participants. For most of them, the time related to *understand* an interaction with Varviz took twice as long than *identifying* it in the trace. Still, we believe we cannot generalize this particular result to other interactions or even other systems, because the suspicious interactions of our tasks was placed in the first third part of the trace, coincidentally. Thus, since the participants started to analyze the trace from the beginning, we believe that the time to identify an interaction is more related to the moment it is called in the execution.

Observation 5: *VarXplorer and Varviz complement each other.*

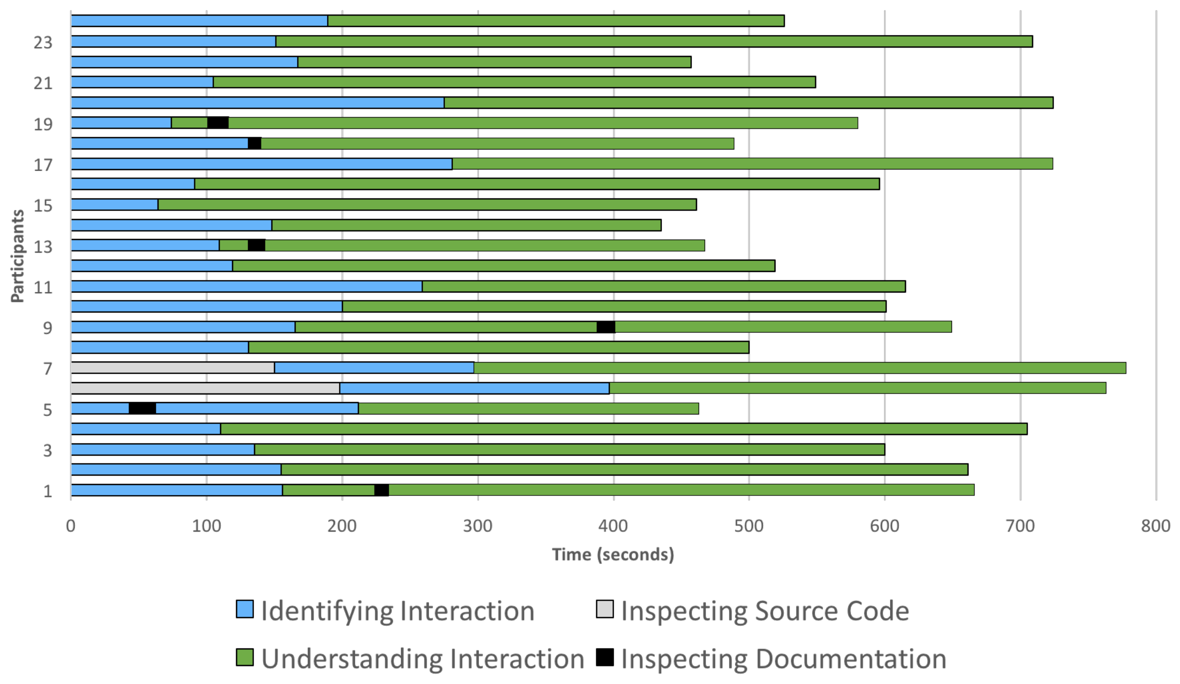
VarXplorer has been discussed as faster and easier to identify and understand an interaction than Varviz. Although Varviz also shows interactions, it has a different purpose. It was designed for understanding faults and program comprehension tasks that involve understanding differences among similar executions [65]. For instance, P3 reported: *"I can use the graph to get an overview on the features that interact, and then I can use the trace to understand the details, see the value of the variables and the flow of execution."*

We observed that Varviz is a valuable strategy to be used after detecting the interactions with VarXplorer. Varviz can be used instead of a standard debugger to look for the cause of an interaction bug. Both tools are Eclipse plug-ins and we believe they may can complement each other in practice.

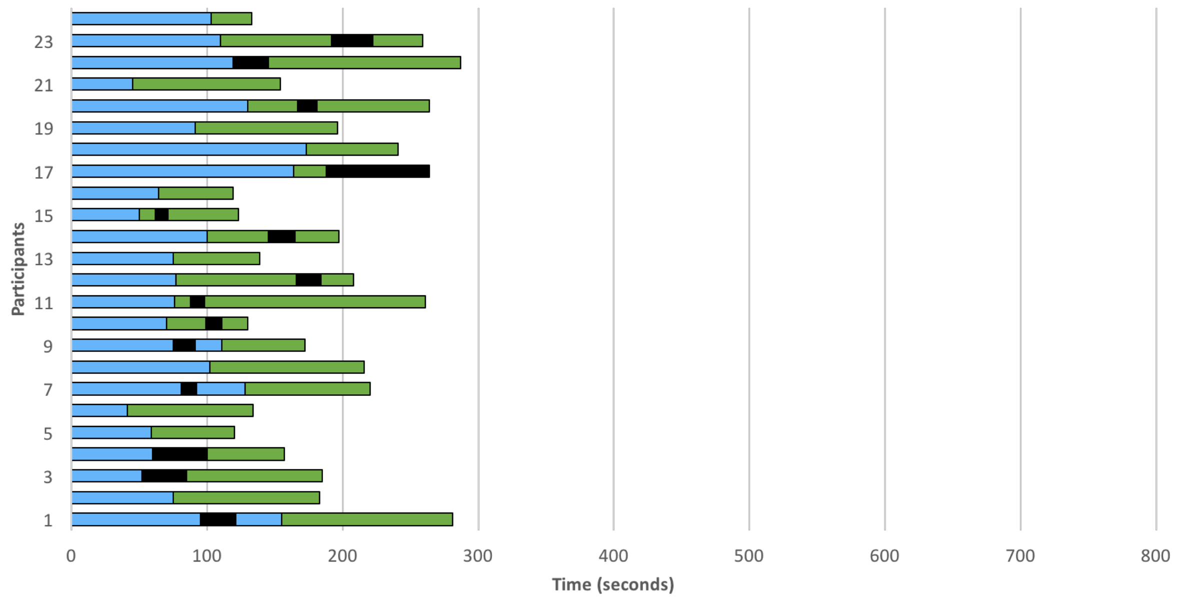
RQ2: The results confirm that the relationships graphically represented as arrows and colors in VarXplorer make the developer work easier and faster. Also, VarXplorer only shows conditional variables, which reduces the amount of information shown to developers.

6.3 THREATS TO VALIDITY

External Validity. We applied our approach to small programs due to the boundaries of an in-lab study; our results may not generalize to larger programs in the wild. However, given that our approach was clearly helpful even in small programs, we argue that is



(a) Varviz Tasks



(b) VarXplorer Tasks

Figure 6.5: Time spent on performing the tasks to Varviz and VarXplorer.

likely helpful for larger systems as it is nearly impossible to detect behavioral interactions without specifications or without specialized tool support [123].

Construct Validity. We did not compare our tool with a standard debugger as baseline, as we believe that the task without specialized tool support (e.g., Varviz) would be too difficult and slow for an in-lab study. Thus, a direct comparison with Varviz, which is specialized to graphically show the execution, is more practical than to compare with a standard debugger. Varviz, at least, shows what happens in the execution when features interact (the trace shows all the possible paths), while using the debugger the developer has no clue where to start looking for interactions. Given that VarXplorer was shown to be significantly faster than Varviz with a large effect size, and that Varviz was shown to outperform the standard debuggger [65], we speculate that comparing VarXplorer to the standard debugger would have produced an even larger effect size. Another threat is related the tools presented to the participants, which had never used them before the experiment. To mitigate this threat, we provided a 10-minutes tutorial before they use each tool, Varviz and VarXplorer, such that all participants be able to use them for the tasks.

Internal Validity. We used 24 participants in our study of which several where students without former experience on interactions (i.e., beginners for this kind of analysis). Experienced programmers for such kind of analysis will perform better for the tasks proposed. However, also experienced programmers will benefit from our tool support as VarXplorer provides them essential information that helps to understand and detect feature interactions. In addition, we used a think-aloud protocol to gain qualitative insights, which may influence the performance of the participants. However, we argue that the influences are similar across the groups and that the differences among the tools are large enough that this influence can be neglected.

Conclusion Validity For this experiment, we used ANOVA [115], a well known statistical techniques. ANOVA is robust to violations of their assumptions, when the groups have the same size. We have four groups with six participants each. In addition, the experiment data is normally distributed and presented a large effect size for our treatments: participants using VarXplorer are more than three times faster than participants using Varviz.

6.4 RELATED WORK

Instead of variability-aware execution, some approaches have performed static analysis to detect interactions [105, 106, 124]. However, despite recent advances, static analysis of systems with high accuracy remains challenging [124, 125]. In contrast, we use a dynamic analysis, variational execution, which is able to analyze large software [49, 50, 126]. Others aim to execute configurations separately, and use symbolic execution to identify interaction problems [127, 47]. Reisner et al. measured the effect of interactions only on control flow using symbolic execution [108], whereas we analyze both control and data flow.

Delta debugging approaches systematically narrow the state difference between a passing run and a failing run [128, 129, 130]. For example, Zeller [128] isolated cause-effect chains for failures. Sumner et al. [129, 130] improved Zeller's work and provided an automatic debugger to precisely align two executions. Conversely, our approach explains differences among many executions. Unlike Delta debuggers, Varviz dynamically tests different executions [65]. However, as far as we know, no work provides explicit information about the relation between features, as we do with suppress and require relationships.

Several approaches work with feature-based specifications to detect interactions. Li et. al [44] present a model checking approach to detect interactions automatically given a group of feature specifications. The approach tests CTL (computation tree logic) properties of features to identify cases in which the specification is violated. Apel et. al [131] also propose a technique to verify whether specifications hold across system configurations. To perform this verification, specifications for intended interactions may be needed, and each feature requires a formal specification of its behavior.

With feature-based specifications, interaction faults can be detected when a feature specification is violated in a configuration. In practice, nevertheless, it is uncommon to create specifications for all features. In general, approaches based on feature specifications present two main drawbacks: (1) from the whole set of features, it is not clear which combinations of features need to be verified and (2) verification tools need precise specifications to check against, information that developers are often reluctant to prepare.

Global specifications only describe properties for all configuration systems, and can thus not describe nuances of intended and unintended interactions to recognize if they affect feature behavior. Generally, it is difficult to find bugs caused by unintended interactions without any specification. Thus, despite their disadvantages, global specifications provide a convenient way of detecting interactions. For that reason, many studies base their approaches on that kind of specifications and focus on exploring the configuration space, such as systematic sampling [38, 39, 40], combinatorial interaction testing [41, 42, 43], model checking [44, 131, 46, 47, 48], and variational execution [49, 50, 51, 52, 126].

6.5 CHAPTER SUMMARY

We conducted a controlled experiment to evaluate how interaction graphs help identify suspicious feature interactions in highly configurable systems. We used two systems very used in the literature once they contain many interactions, Elevator and Telephone. Then, we compared VarXplorer with another tool, Varviz, and we found that VarXplorer is on average 3 times faster than Varviz. Next chapter, we present a complementary study that explores other aspects of VarXplorer, namely the iterative process and feature-interaction specifications.

Chapter 7

EXPLORATORY STUDY: AN ANALYSIS ON VARXPLORER ITERATIONS

VarXplorer is an approach to support developers identifying feature interaction problems. Our detection strategy involves an iterative process of detecting and documenting evaluated interactions. In Figure 5.1, we presented an overview of our approach and how it incrementally supports the analysis of feature interactions. Given a configurable system, we execute test cases (system inputs) looking for feature interactions. Users start with one test case and based on the feature interaction graph, they explore which interactions are either problematic or benign. The graph provides a visualization of which features interact, besides presenting relationships and variables.

We present this feature-interaction graph to developers for manual inspection. From the graph, they can select an interaction, which connects two features, to allow benign interactions or mark others as suspicious. From this process, developers automatically create feature interaction specifications that are documented and used for future test cases.

This exploratory study is complementary to the one presented in the previous chapter. Instead of evaluating just the graph, in this second study we explore the entire approach to investigate how the iterative and interactive approach may support the discovery of suspicious interactions. The previous study aimed at understanding how feature-interaction graphs can help developers identify suspicious interactions. It focused on the understanding of the graph and how fast users could identify problematic interactions compared to the state-of-the-art tool. Thus, it did not discuss about the iterative approach and feature-interaction specifications.

The objective of this current study is to analyze how feature-interaction specifications can be iteratively used to reduce the effort of identifying and judging interactions from a set of test cases. This chapter consists of six major sections:

Section 7.1 presents the research questions of our study;

Section 7.2 describes the subject system, as well as its features and implementation;

Section 7.3 presents the design of our study, overview and procedure;

Section 7.4 discusses and analyzes the results of our study, and presents an analysis of the order of tests execution.

Section 7.5 shows the lessons learned after the exploratory study; and

Section 7.6 discusses the threats to validity;

Section 7.7 concludes this chapter.

7.1 RESEARCH QUESTION (RQ)

In our process, with the support of user interaction, the feature-interaction graph is automatically refined by removing benign interactions over test cases. When using the tool, the user can right click on interactions and mark them as benign or problematic. This refinement is supported through a *feature interaction specification language*. Interaction specifications aim to point out the existence of an interaction between two features, without the need of a formal behavior specification.

Hence, interactions marked as benign are removed of the graph. When executing other test cases, the developer does not see that benign interactions again in future graphs. In case of bugs, the user may want to fix the problem directly in the source code and also mark those interactions as suspicious in the graph. The feature-interaction language helps developers to either allow or forbid interactions in a configurable system. The goal is to incrementally remove intended interactions in order to make the analysis of tests less complicated and faster, focusing on newly unintended interactions. This exploratory study aims to answer the following research question:

RQ: How do the iterative process on individual test cases reduce the complexity of identifying interactions?

Iterativeness means the potential to optimize the feature-interaction detection through short iterations in sequence, and each iteration has a self-contained program scenario composed of one test case analysis. In the iterative process, a single test case is analyzed at a time, which produces one feature-interaction graph with all possible interactions and relationships among the features for the given scenario.

For this question, we are also interested in understanding how much effort we save using interaction specifications when executing test cases. Feature-interaction specifications can be used to inform which interactions are benign or problematic in a test case. Then, they can be applied over other test cases to remove interactions that are already known as benign to the program, besides highlighting problematic interactions that may occur in different tests.

The effort to analyze interactions is related to the size of the graph: the bigger the graph is, the more interactions the developer has to check whether they are suspicious or benign. Thus, we also compare the size of graphs without applying specifications (complete graph) versus the reduced graph, when known interactions are removed. In this way, we measure how many interactions the developer are not seeing again during the iterative process.

7.2 SUBJECT SYSTEM

We conducted an exploratory study on the basis of the RiSE Event SPL [132]. This SPL is dedicated to support organizers of a conference. It comprises papers submission in conferences, journals, and related events, and its management, including the control over the review life-cycle as well as the management of activities (workshops, tutorials, panels), users (speakers, organizers, reviewers), registrations, payments and certificates. RiSE Event SPL was based on the main features found on largely used conference management systems, such as: EasyChair[1], JEMS[2] and CyberChair[3]. The SPL is able to generate different products to different conferences styles.

RiSE Event SPL was developed using Java language, Model-View-Controller (MVC) architectural pattern and MySQL database. The SPL was initially annotated for compile time. The conditional compilation was employed to isolate each functional property code and a build file[4] was used to derivate SPL products.

For this study, we adapted the SPL to runtime annotation, using JPF annotations (Java library for runtime code annotations). We replaced each conditional annotation by a similar annotation that can be read by VarXplorer. We maintained the same expression created by the SPL developers for the conditional compilation. Therefore, the replacements are equivalent in terms of logical meaning. Figure 7.1 shows an example of a replacement. An instance of the `PaymentRepository` *class* is created under the condition defined by the presence of at least one of the following features: `PaymentCash`, `PaymentDeposit`, and `PaymentCard`. As previoulsy explained in Chapter 5, for the runtime analysis, features are annotated as conditional values in the program.

RiSE Event SPL was chosen to this study because it is written in Java, contains many features, the documentation is available, and the developers are most of the time available to discuss the results. They developed the SPL without using any kind of systematic testing analysis.

7.2.1 Features

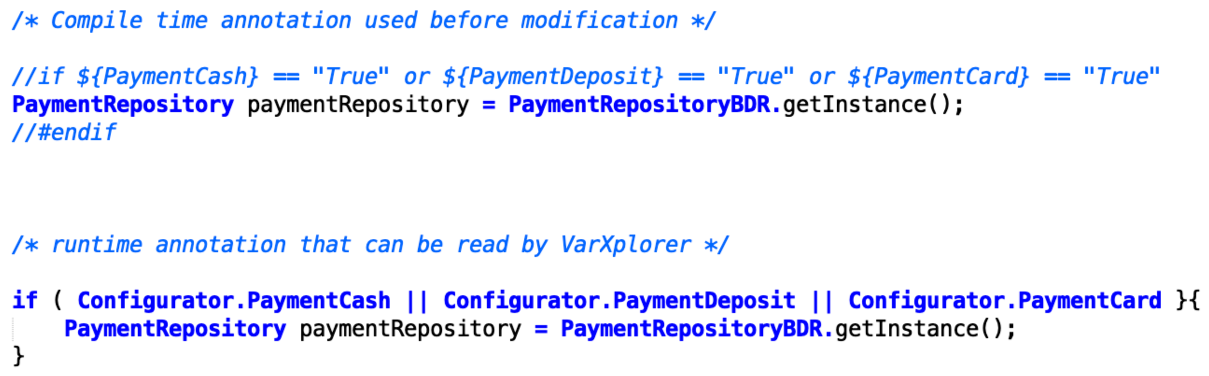
The version of the RiSE Event SPL used in this study is composed of 20 functional features, 26.457 Lines of Code, 1493 Methods and 496 Classes. All features have their corresponding code delimited by runtime annotations. The features are named as follows: *Event, User, Reviewer, Speaker, Author, Activity, ActivityTutorial, ActivityWorkshop, Registration, RegistrationSpeakerActivity, RegistrationUserActivity, CompleteSubmission, PartialSubmission, Review, Payment, PaymentCash, PaymentDeposit, PaymentCard, Assignment, and ConflictOfInterest.* Figure 7.2 shows the feature model of the SPL developed before and independent of our study, by other developers [132].

[1]www.easychair.org/

[2]jems.sbc.org.br

[3]www.borbala.com/cyberchair/

[4]https://ant.apache.org



```
/* Compile time annotation used before modification */

//if ${PaymentCash} == "True" or ${PaymentDeposit} == "True" or ${PaymentCard} == "True"
PaymentRepository paymentRepository = PaymentRepositoryBDR.getInstance();
//#endif



/* runtime annotation that can be read by VarXplorer */

if ( Configurator.PaymentCash || Configurator.PaymentDeposit || Configurator.PaymentCard }{
    PaymentRepository paymentRepository = PaymentRepositoryBDR.getInstance();
}
```

Figure 7.1: The same code excerpt annotated for compile time and runtime.

7.3 EXPERIMENTAL STUDY DESIGN

This section presents details of the experiment, how it was defined, executed and analyzed.

7.3.1 Study overview

The iterative process proposed by VarXplorer consists of executing test cases incrementally. The process starts by running a small test, which tends to generate a small graph. Then, we incrementally increase the test complexity (increasing number of features and test activities) until we have tested all system functionalities.

We execute each test using VarXplorer and analyze the feature-interaction graph. The analysis consists of the execution of a test case one at a time. The user checks each interaction and marks it as either benign or suspicious (right clicking on edges of the graph and choosing an option - either *allow* or *forbid* the interaction). When an user marks the graph, it automatically creates feature-interaction specifications to document interactions defined as benign and problematic. Those specifications, as described in Section 5.3, differs from global and feature-based specifications because they do not represent a formal description of the feature and system behavior. Instead, it provides a lightweight strategy to indicate that there is an interaction between two features.

VarXplorer also shows partial interactions, the ones related to variables, which can be analyzed separately for each variable. In case of finding problematic interactions in a graph, we first need to fix the problem at source code and then, run the test again to check how the features are interacting. We repeat this process until we believe that the test does not contain any other suspicious interaction.

VarXplorer proposes to run test cases in sequence, in which the next test case is run after the analysis of the previous one ends. The analysis of a given test includes: (i) understanding each interaction of the graph, and; (ii) having all bugs fixed, if any. This process is iterative and incremental until we have executed all tests from the test case suite. The tests were executed using the same laptop (2.3GHz i5, 16 GB DDR3) and Eclipse IDE (Oxygen version).

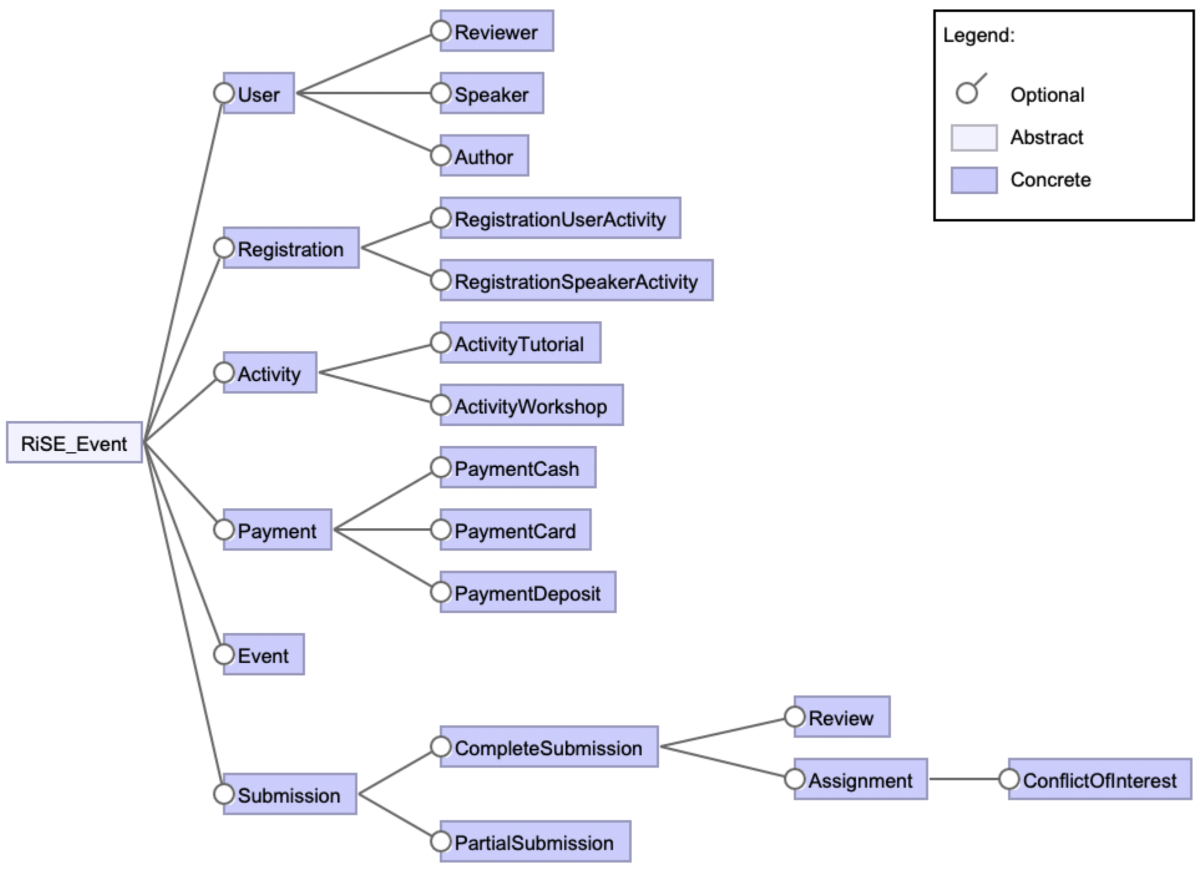


Figure 7.2: Feature model of the RiSE Event SPL.

7.3.2 Design

Before running the actual experiment, we performed a set of preparatory steps, as Table 7.1 shows. First, we had a practical session with the SPL developers to configure the Eclipse environment and database. Then, we adapted the SPL annotations to runtime code annotations. The session to modify the annotations took 8 hours split across 2 days.

The RiSE Event SPL project did not have any test cases available for this study. Therefore, the process to create the test case suite was supported by the original SPL developers. We performed a brainstorming session to discuss the possible scenarios and how the tests would be built. At the end of the session, we defined each scenario in natural language. The brainstorm session took around 2 hours. Then, we implemented the tests in Java and validated them with the developers to certify whether the tests would achieve the proposed goals (features coverage). The test case suite was developed in about 8 hours, and the validation session took 2 hours.

After the initial phase, *preprocess*, the actual experiment was conducted. We created a test suite with 15 tests. First, we executed tests 1 to 7 and performed a verification session with the RiSE Event SPL developers to present the information gathered. Interactions, problems, causes and solutions (fixes at source code) were presented to be discussed. The

Table 7.1: Experiment Design

Preprocess			
#	**Activities**	**Objective**	**Involved Subjects**
1	Practical Session	Adapt annotations & configure environment	SPL developers and VarXplorer expert
2	Brainstorm	Define scenarios	SPL developers and VarXplorer expert
3	Implementation	Tests development	VarXplorer expert
4	Validation	Tests validation	SPL developers and VarXplorer expert
Actual Experiment			
1	Tests 1 to 7	Execution and analysis of tests 1 to 7	VarXplorer expert
2	Verification	Validation of bugs and interactions found	SPL developers and VarXplorer expert
3	Tests 8 to 15	Execution and analysis of tests 8 to 15	VarXplorer expert
4	Verification	Validation of bugs and interactions found	SPL developers and VarXplorer expert

same process was conducted for tests 8 to 15: we first executed and analyzed the tests and then validated them with the SPL developers. This experiment took 2 weeks. Table 7.2 presents the list of tests.

In addition, after the analysis in sequence from test 1 to test 15, we created and run three sets of the same tests randomly organized. This analysis checked whether the order of the tests have any influence in the results.

7.3.3 Procedure

Before running the tests, we also created a test harness, we filled out the SPL database with some initial information. We created 4 events, 4 authors, 5 users, 4 activities, 2 speakers, 3 reviewers, 4 reviews, 4 submissions, 3 payments, 2 registrations, and 3 assignments. The test cases manage this information in the database, searching, changing or creating new data. Figure 7.3 illustrates this information contained in the database before running the test suite.

We implemented test cases that comprehend possible use scenarios. We started with small scenarios composed of few features and incrementally increased the tests in terms of complexity and number of features. For example, one test creates events, and the next one creates and updates events. The saturation was achieved when we had tested at least all features and the creation of new test cases did not generate any new interaction. Thus, our set of test cases is composed of scenarios, extensions of other scenarios and new scenarios. In other words, the test suite has a mix of tests with the same features, but different inputs; tests with features not tested in previous tests; and tests with a mix of non-tested features and features already tested, as Table 7.2 shows.

When executing a given test case, the variational execution tests all combinations among the features of that test in one single execution. For instance, if we consider a test with 3 features (A, B, and C), a single execution is able to test the program and all the features combination, enabling and disabling each feature. This process, earlier presented

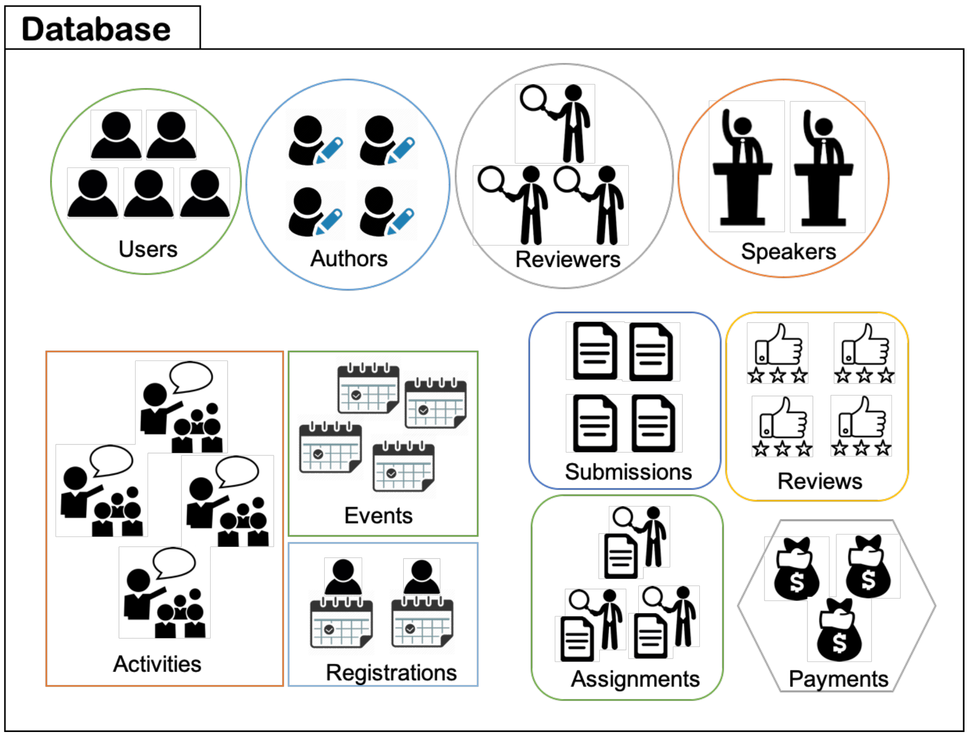


Figure 7.3: Database used for the tests.

in Section 4.5 in this Thesis, is similar to execute the same test case 7 times, one for each possibility among 3 features (i.e., A, B, C, AB, AC, BC, and ABC). Instead of creating 7 tests, the variational execution tests all the 7 possibilities in a single execution.

Due to the ability of exhaustively executing a test case over all configurations of a software product sharing redundancies of the executions, the test suite does not need to contain one test to each configuration. Consequently, the variational execution reduces the need for additional tests. In this way, the basis for designing tests to run under variational execution is to consider different user scenarios, which should simulate how the user would use that system. We used the white-box technique to create system tests. The goal of *system testing* is to run the system from the point of view of its end user [133]. The scenarios contain different features and the set of tests aims to evaluate all the functionalities of the system. For example, T6 registers new reviewers; the following test extends T6 by registering a new submission; and T11 tests other functionality, it registers an user in an activity, both from the database.

The analysis procedure consists of running each test separately. Figure 7.4 shows the flow chart of this process. When a test is executed with VarXplorer, the corresponding feature-interaction graph is created. To analyze interactions, users have to check the graph and judge them as either benign or suspicious. In case the users identify suspicious interactions, we suggest them to look for the causes of the problem and fix them at source code before running the next test case. Then, they have to run the test again and check the graph after the fix to guarantee the problem has been fixed. At this point, the graph created after the fix should not contain the same interactions of the former

Table 7.2: Details of the Test Case Suite

Test	**F**	**Description**
T1	4	It creates and updates events, and updates activities
T2	4	In addition to 1st test, it also creates activities
T3	3	It adds a registered speaker to a registered activity
T4	4	In addition to 3rd test, it creates new speakers
T5	4	In addition to 4th test, it adds new speaker to a registered activity
T6	2	It creates reviewers
T7	5	In addition to 6th test, creates a new submission
T8	6	In addition to 7th test, attributes new submissions to users
T9	7	In addition to 8th test, attributes submissions to authors
T10	7	In addition to 9th test, creates reviews to submissions
T11	4	It registers an user in an activity
T12	8	In addition to 12th test, registers a payment
T13	6	It creates assignments to different reviewers
T14	7	In addition to 13th test, checks conflicts between authors and reviewers
T15	8	In addition to 15th test, send messages to reviewers

F: number of features;

graph. Since the source code may have changed, the behaviour of the program changed as well. The graph may contain new interactions, and others related to the bug may not happen anymore. Thus, this graph need to be checked again, and the process restarts: the interactions of this new graph should be checked and judged until all interactions have been understood.

7.4 RESULTS

This section presents the results of our study and discusses implications. We next present our exploratory analysis to answer our research question on how the iterative process on individual tests reduces the complexity of identifying interactions.

We executed the tests sequentially, one by one, from T1 to T15. Table 7.3 shows the type of interactions found for each test, besides variables identified and time of analysis. The tests range from 2 to 8 features each. The smallest graph (T6) has 2 interactions, and the biggest one (T12) has 32 interactions. They are composed of *require* and *suppress* relationships, and conditional variables.[5] For instance, T15 presented 29 interactions in total, but since the graph was simplified based on specifications built from T1 to T14, it was reduced to 17 interactions, 16 out of them are *require* interactions and 1 is a *suppress* interaction.

The test suite has a total of 431 conditional variables and 149 different interactions. 118 are *require* interactions and 31 are *suppress* interactions, scattered over 15 tests. 11 *suppress* interactions and 6 *require* interactions presented problems. The system used in this study has 20 features, in which each feature interacts at least with 3 other features for this test suite. The least interacting features are `Event`, `ActivityTutorial`, and

[5]Variables that depend of at least 2 features.

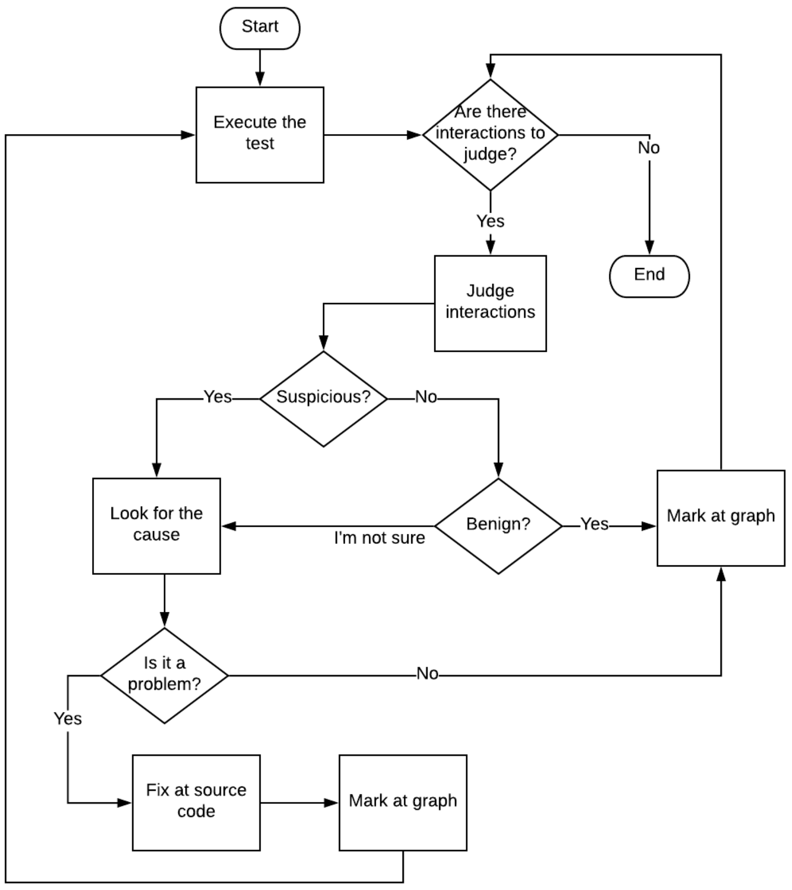


Figure 7.4: Flow chart of the process to analyze interactions of a test case.

`ActivityWorkshop`. On the other hand, the features `Activity` and `User` are the ones that interact the most, with 10 features each.

The first test (T1) creates new events, updates recently created events, and updates the price of an activity registered in the database. Figure 7.5a shows the graph presented to the user after executing T1. There are 4 interacting features, which interact with at least with 1 other feature. Based on the domain knowledge, T1 should only contain interactions between `Activity` and `Event`. Nonetheless, the graph presents interactions with two other features: `ActivityTutorial` and `ActivityWorkshop`. These two last features should neither has effect nor interact with others for that test. In addition, the feature `Activity` should not suppress any other feature of the system, as the graph shows. As a result of the analysis, we recognized those interactions as suspicious and with high chances of representing a bug in the system.

Although we were able to identify interactions as suspicious based on the feature-interaction graph, it was not enough to understand what triggered that *suppress* interactions. The graph was proposed to support and provide a fast identification of problematic interactions, and the source code should still be used to solve the problem. Thus, to un-

Table 7.3: Interactions type, variables and time of analysis

Test	Interactions				Time
	# Require	**# Suppress**	**Total**	**Variables involved**	
T1	2	4	6 (6)	17	8 hours
T2	6	1	7 (7)	7	1 hour
T3	4	0	4 (4)	6	30 min
T4	7	0	7 (8)	43	20 min
T5	10	4	14 (15)	56	20 min
T6	2	0	2 (2)	3	5 min
T7	4	2	6 (8)	32	10 min
T8	10	2	12 (14)	32	10 min
T9	4	2	6 (15)	22	5 min
T10	6	2	8 (19)	30	5 min
T11	12	0	12 (12)	30	15 min
T12	18	4	22 (32)	94	20 min
T13	8	1	9 (9)	13	5 min
T14	9	2	11 (20)	17	15 min
T15	16	1	17 (29)	29	10 min

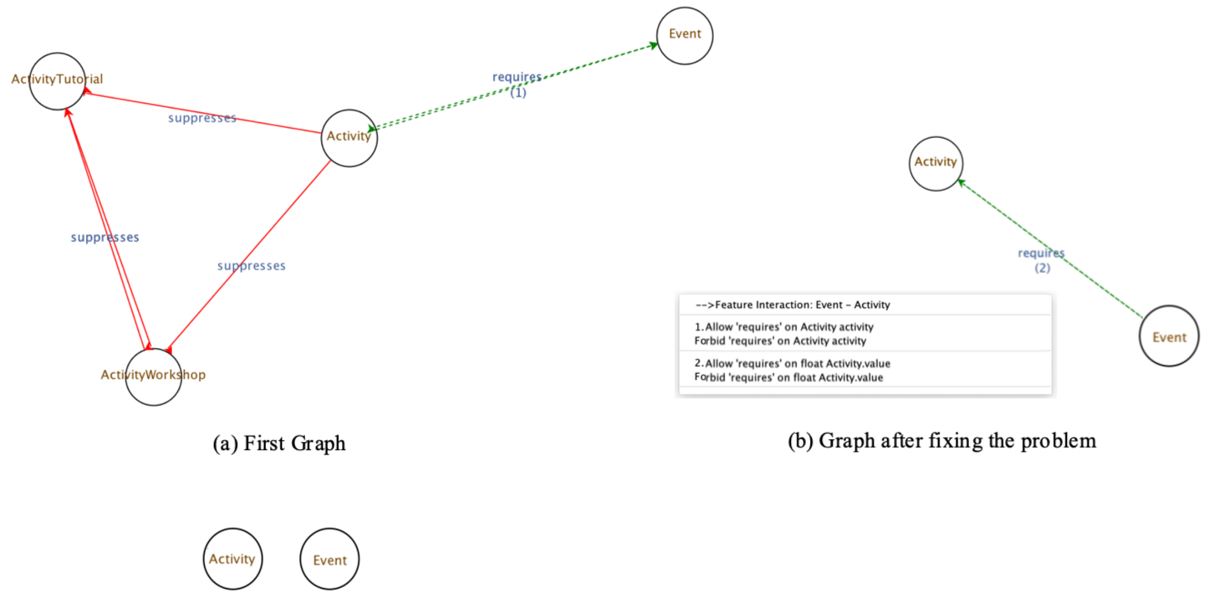
#Require: number of interactions of *require* type; **#Suppress**: number of interactions of *suppress* type; **Total**: x (y), x means number of interactions shown in the graph after applying specifications, and y means number of interactions when no specification is applied; **Variables involved**: number of conditional variables present on the interactions of that test case; **Time**: Time spent to analyze each graph, judge as benign or problematic, and fix the problems at source code (if any).

derstand the cause of the suspicious interactions of T1, we analyzed the execution trace, the source code and also used the Eclipse debugging tool. For this first test, we used all the available resources to look for the problem, which was a wrong expression used in one annotation. Because of the wrong annotation, the program was calling unnecessarily routines and interactions.

Since T1 was the first test case of our analysis, it took longer to be understood than the other tests. After finding the problem and fixing it at source code, we came back to the graph and marked the suspicious interactions as *forbid*. In order to confirm that those interactions had been fixed, we ran the test case again. Figure 7.5b shows this graph.[6] After fixing the problem at source code, the graph only shows one interaction: feature `Event` *requires* feature `Activity` related to 2 variables (*activity and value*). According to our analysis of the source code and domain, this interaction behaves as expected and then, it was marked as benign in the graph. Benign interactions are automatically removed. Figure 7.5c shows the same graph presenting that the interaction between `Event` and `Activity` has been removed because it is benign.

The entire process to analyze T1 took over 8 hours. Figure 7.6 shows the set of specifications automatically created after finishing the analysis of T1: two interactions marked as *forbid* and one interaction that includes two variables marked as *allow*. At the end of the analysis of all tests, the set of specifications should contain all the specifications

[6]The steps of this process is shown on Figure 7.4.



ActivityTutorial
suppresses
Activity
requires
(1)
Event
suppresses
suppresses
ActivityWorkshop
Activity
requires
(2)
Event
-->Feature Interaction: Event - Activity
1. Allow 'requires' on Activity activity
Forbid 'requires' on Activity activity
2. Allow 'requires' on float Activity.value
Forbid 'requires' on float Activity.value
Activity
Event

(a) First Graph

(b) Graph after fixing the problem

(c) After applying specifications

Figure 7.5: Graphs generated for the analysis of T1.

created from T1 until T15. The set starts with the specifications automatically created for T1 and increases incrementally until the analysis of the last test case, T15. As a result, at the end of the last test, we created a total of 286 specifications. This number is high because one interaction is created to each variable analyzed by the user. Appendix C shows the final file with all specifications.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<system name="RiSE Event SPL">
    <specification type="Forbid"><suppress from="Activity" to="ActivityTutorial"/></specification>
    <specification type="Forbid"><suppress from="Activity" to="ActivityWorkshop"/></specification>
    <specification type="Allow"><require from="Event" to="Activity"><var name="Activity activity"/></require></specification>
    <specification type="Allow"><require from="Event" to="Activity"><var name="float Activity.value"/></require></specification>
</system>
```

Figure 7.6: Specifications automatically created after the analysis of T1.

An specification refers to either the whole interaction (*total relationship*) or a single variable of an interaction (*partial relationship*). For example, the two first specifications of Figure 7.6 forbid any interaction between that pairs of features, regardless the involved variables, they are all forbidden. On the other hand, the last two specifications allow interactions between features `Event` and `Activity` only related to the variables *activity* and *value*. Since other variables of this interaction are not mentioned, if different variables appear in future test cases, they will be shown in the graph to be judged by the user again.

From the second test case, VarXplorer recognizes that new specifications were created and applies them to reduce the graphs. This process cleans all benign interactions in case they are repeated over the tests. For example, T2 is an extension of T1. It has not only more features and more interactions, but also has that same interaction between `Event`

Table 7.4: Test Cases: Problems identified and percentage of cleaning

Test	Interactions	Found and Fixed Bugs		Spec Cleaned from others	% of Reduction	
		Require	Suppress		Int.	Vars
T1	6	0	2	0	0,00%	0,00%
T2	7	0	1	5v (1i)	0,00%	45.45%
T3	4	0	0	0	0,00%	0,00%
T4	7	0	0	1i, 9v (3i)	12.5%	4.92%
T5	14	0	4	1i, 51v (8i)	6.66%	47.66%
T6	2	0	0	0	0,00%	0,00%
T7	6	0	1	2i, 3v (2i)	25,00%	27.27%
T8	12	0	0	2i, 32v (7i)	14.29%	47.76%
T9	6	0	0	9i, 66v (13i)	60,00%	70.21%
T10	8	0	0	11i, 58v (16i)	57.89%	44.96%
T11	12	0	0	0	0,00%	0,00%
T12	22	6	2	10i, 30v (12i)	31.25%	31.58%
T13	9	0	0	4i, 6v (4i)	30.77%	31.58%
T14	11	0	1	9i, 17v (11i)	45,00%	53.13%
T15	17	0	0	12i, 25v (14i)	41.38%	50,00%

Interactions: number of interactions presented in the graph with applied specifications; **Require**: number of problematic interactions of type *require*; **Suppress**: number of problematic interactions of type *suppress*; **Spec Cleaned from Others**: number of specifications removed from the graph after applying the set of specifications; in which [xi, yv (zi)] means x interactions removed and y variables from z different interactions; **Int**: percentage of interactions removed from the graph; **Vars**: percentage of variables removed from the graph.

and `Activity`, removed by the specifications. Thus, VarXplorer reduces the number of interactions that the user has to judge.

During the analysis of T2, one interaction was identified as suspicious. After the analysis of the source code, it was confirmed as a problem: a workshop activity was not created as it should because another feature blocked its effect, `ActivityTutorial` *suppressed* `ActivityWorkshop`. The problem was caused by a wrongly nested code. Table 7.4 shows that 6 out of 15 tests presented problems, and 17 interactions were marked as suspicious.

The feature-interaction problems found in the RiSE Event SPL were validated with the SPL developers twice during the test process, as Table 7.1 shows. All problems were confirmed and none of them were refuted during the verification sessions. RiSE Event SPL presented 5 types of problems:

- **Bad Annotation**. The expression used in the annotation is incorrect. It creates wrong interactions and may generate malformed products.

- **Wrong Object**. An object used in place of another. The program was implemented based on MVC pattern. A method was copied from another class and part of it was maintained as it should not.

- **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.
- **Spread Code**. Piece of code spread over different features leading to unnecessary dependencies and problems related to code modularity.

We did not find any problem that led the program to a crash. However, some problems, such as *wrong object*, *conditional statement*, and *misplaced overwrite* may lead to a wrong behaviour. For example, in T7, the user is obliged to fill all paper information before submitting it, although the program should allow partial submissions. Other problems provide unnecessary dependency and impair software maintenance/evolution.

From the analysis of T2, VarXplorer used the specifications created during the analysis of T1 to remove benign interactions of the next tests. Table 7.3 and Table 7.4 show the number of interactions and variables cleaned during the testing process from T1 to T15. For example, the graph of T10 presents 19 interactions but 11 were removed from specifications. The user had to judge only the 8 interactions showed in the reduced graph. In addition, at the same test, 58 variables of 16 different interactions (*11i, 58v (16i)*) were also removed from specifications. They all appeared in previous tests (T1 - T9) and were previously marked as benign. VarXplorer also allows the user to see both graphs, the reduced and the complete one. For T10, the user had 57.89% less interactions and 45% less variables to analyze when using the specifications provided by VarXplorer.

Figures 7.7 and 7.8 show both graphs of T10, complete and reduced, respectively. The reduced graph has less interactions and, thus, it is faster and easier to analyze compared to the complete one. Hence, the user need to analyze only the new interactions, which appeared for the first time in this graph. For example, all interactions related to `User`, `InsertAuthor`, and `Activity` were removed because they were judged as benign during the analysis of T9. Appendix C.2 shows all the graphs used during the analysis of the 15 test cases. It includes the reduced graphs, intermediary graphs generated after fixing bugs, and final graphs generated after judging interactions as allow or forbid. Although the goal of this study is to explore specifications using the reduced graphs to do the analysis, we also show the complete graph in the Appendix C.2 to serve as comparison to the reader.

In general, by using the iterative process and the specifications, we had 45% less interactions to judge in average; and 50% less variables, considering the graphs that presented reductions. Graphs with 0% of reduction (T1, T3, T6, and T11) represent new scenarios containing features not tested in previous tests during the analysis. Thus, they have interactions not seen in previous tests.

7.4.1 Analysis of Order Influence

In this study, we are investigating the iterative process on individual test cases and how it reduces the complexity of identifying interactions. For the iterative process, we created the test suite incrementally, increasing the number of features and scenarios complexity. The tests were executed sequentially and ordered from the smallest to the largest graph, T1 to T15. Small graphs have less interactions and are easier to analyze compared to large

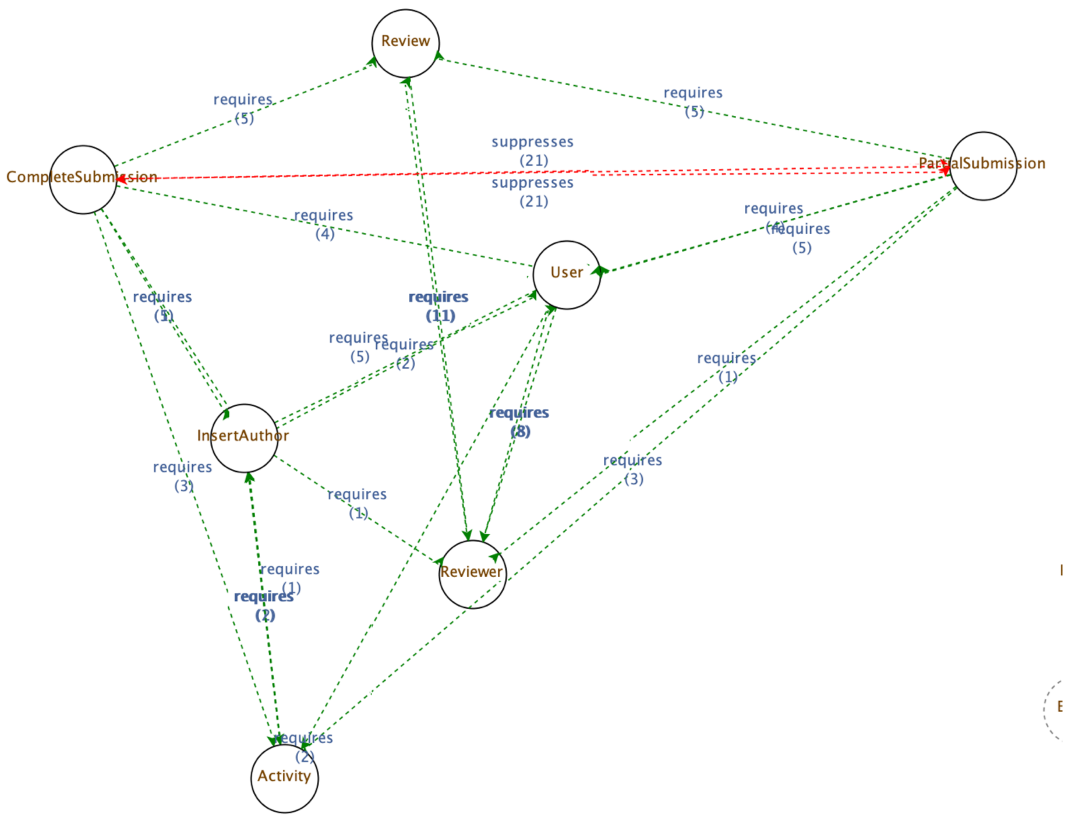


Figure 7.7: Complete graph of T10.

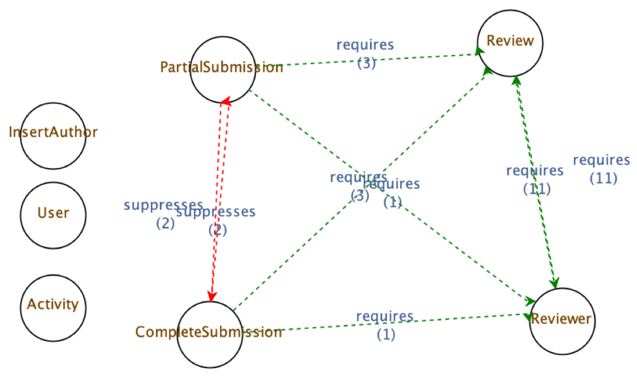


Figure 7.8: Reduced graph of T10.

ones. For example, we started with T1 with 4 features and 6 interactions to judge, and finished with T15 with 8 features and 17 interactions to analyze, because 12 interactions were removed during the process.

After the analysis of all tests and validation with the SPL developers, we randomly changed the order of the tests and created three different sets of tests. The analysis was conducted to check whether the order in which the tests were executed had any effect in the iterative study. Table 7.5 shows these sets and the additional effort needed for each analysis. The value *zero* in Table 7.5 means that no additional effort was used to judge the graph, i.e., either it has the same number or less interactions than the same graph of the ordered list.

For the set of tests analyzed in an orderly manner, each subsequent test is reduced because of benign interactions that have been identified in previous tests. However, when we change the order, we do not guarantee that small tests come first and the analysis may have a high initial effort. For example, T8 was the first test to be analyzed in the first random set (1st R set) of Table 7.5. For this analysis, T8 has 2 interactions and 25 more variables (+2i 25v) compared to the original test showed in Table 7.3. This happened because when the tests were executed in an orderly manner, T5, T6 and T7 were executed before T8. The effort to analyze T8 in the random set was higher compared to the ordered execution. The tests are incremental and T5 to T8 have a similar scenario, but they grow in the number of features. Thus, the effort to analyze T8 this time was higher than when executed in an orderly manner.

Although the initial effort may be higher, the overall effort is the same for all sets of tests, regardless the order. The effort here is related to the number of interactions to analyze: the amount of unique interactions and variables in each set is the same. All the tests presented 149 different interactions and 431 unique conditional variables. We did not find any new interaction, variable, or problem due to variations on the order of execution. Hence, the number of unique interactions did not change, independently of the order of execution.

When large graphs are executed first and out of order, users have much more information to check. The number of removed interactions is reduced and the complexity of the analysis increases, the user see more interactions in a single graph. Conversely, analyzing small graphs first, when they get to analyze a large graph, it is likely that many interactions have been removed during the small graphs analysis. VarXplorer proposes to execute tests in an orderly way, which decreases the complexity of the feature-interaction analysis.

7.5 LESSONS LEARNED

For this exploratory study, we used VarXplorer to find feature-interaction problems in the RiSE Event SPL. We executed a test suite with 15 different tests and scenarios. As the test cases grow and features are repeated among tests, many interactions are repeated and, thus, they are no longer showed. The graph then shows only the new interactions, getting smaller and simpler to interpret. For example, the graph of test 9 was reduced by over 61% less interactions when using the iterative process and specifications. It was

Table 7.5: Three random samples of the test suite

1st R set	**Additional Effort**	**2nd R set**	**Additional Effort**	**3rd R set**	**Additional Effort**
T8	+2i 25 v	T10	+11i 58v	T8	+2i 32v
T14	+10i 15v	T6	0	T3	0
T3	0	T9	0	T12	+10i 30v
T10	+2i 10v	T3	0	T5	+4v
T2	+7v	T14	+9i 17v	T13	+4i 6v
T13	0	T2	+9v	T15	+5i 12v
T7	0	T15	0	T10	+9i 15v
T5	+7v	T7	0	T1	0
T9	0	T5	+36v	T6	0
T1	0	T11	0	T9	0
T15	0	T1	0	T11	0
T4	0	T4	0	T14	0
T11	0	T12	0	T2	0
T6	0	T13	0	T4	0
T12	0	T8	0	T7	0

R set: random set.

cleaned of 11 interactions and 58 variables.

Appendix C.2 shows how the graphs would be if they were not reduced (complete graphs), reduced graphs, intermediary graphs, and final graphs. Intermediary graphs are the ones generated after fixing problems at source code in order to solve suspicious interactions. Those suspicious interactions may generate additional interactions that automatically disappear when the problem is fixed. To avoid unnecessary work of analyzing interactions created from bugs, we start the analysis with *suppress* interactions first. Most of the problems we found were related to *suppress* interactions. Only 1 graph out of 15 presented suspicious cases related to *require* interactions, (T12, as Table 7.4 shows). All the suspicious cases were double checked with the developers.

6 out of 15 tests presented feature-interaction problems. However, the number of problematic interactions was much lower than the number of benign interactions. 17 interactions out of 149 were problematic, which represents less than 12% of the total interactions. As expected, most of the interactions that appeared in the graphs were benign. Although the RiSE Event SPL had never been tested before this study, it did not present a high number of problems, nor problems that cause the program to crash.

Most of the problems we found were related to lack of source code modularity and incorrect implementation, such as, wrong variable overwrite, misalignment of *if* statements, and the instantiation of wrong objects. When those problems only appear in the combination of features, they are harder to be identified by common strategies, because they need to test *all* interactions. VarXplorer provides a dynamic inspection process to detect any feature interactions problem that causes differences in control flow and data flow of the system. From the insights of this study, we believe that the use of good programming practices and design patterns is likely to avoid most of the feature interaction problems.

RQ: The results confirm that the analysis of individual and ordered test cases and the use of reduced graphs (with automatic generated specifications) led to a reduction of 45% less interactions to judge in median; and 50% less variables when compared to complete graphs that were not reduced.

7.6 THREATS TO VALIDITY

Construct Validity. For the first test cases, it took 1 hour or more to judge interactions as either benign or problematic, mainly because in the two first tests, we found suspicious interactions, and we were not familiar enough with the program yet. Although we have used the SPL source code for over two weeks to develop the test cases and replace the annotations, we were not the one responsible for the development of the program. However, as we executed the next tests, this time of analysis was drastically reduced to few minutes. For example, for T1, we screened the program for hours, until we realized that the problem was in the annotations. With the exception of T1 and T2, most of the tests took less than 20 minutes to be analyzed, as Table 7.3 shows. This reduction is likely due to the knowledge about the system, which increases as we analyze the next test cases.

The exploratory study was performed by one person using a third party system. However, before the tests execution, the person had unlimited access to the source code for over two weeks. During those weeks, several practical sessions were performed with the presence of the system developers. In addition, the system developers were available most of the time to answer questions in person, email or chat. Other meetings were also realized during the execution of the tests to present the intermediary results and validate the identified feature-interaction problems. The solutions to the problems at source code were also discussed with them.

Internal Validity. There is no systematic process to create a test suite to discover feature interactions with variational execution. The test suite of this study was defined with the developers, and the saturation was achieved when variations in the defined scenarios did not bring new interactions. Individually, the test cases we used combine multiple activities in a single test case. They covered all features and each feature interacted with at least 3 other features. Besides that, we presented the results to the developers and they did not miss any interaction.

Hence, the test suite has reasonable, but not complete, coverage. We cannot guarantee that we covered all possible interactions among all features. Although we potentially may miss interactions that occur only with other specific scenarios, we executed the program with representative inputs, which covers all functionalities.

External Validity. RiSE Event SPL has never been tested before, and has no test case implemented before this study. We implemented 15 test cases and found 11 suspicious interactions and all of them were solved at source code. We do not have any basis of comparison related to other testing approach to check their findings related to the RiSE Event SPL. However, VarXplorer is one of the few tools able to test and identify problems at runtime. Developers were much faster at identifying interaction problems when using

VarXplorer than with another similar tool, as shown on the controlled experiment of Chapter 6.

The graphs used during this study present from 2 to 22 interactions, and the largest graph has 8 features. In a real setting, many more features can interact and the graphs can become too large. Since big graphs may contain many edges, they are more difficult to analyze. To avoid this, we can divide a long test case into smallest tests. Although this process creates more graphs to analyze, they present less interactions and facilitates the process of identifying suspicious interactions.

The RiSE Event SPL presents 20 features and more than 26.000 lines of code. Despite we have used 15 test cases that comprehend different scenarios, the study was carried out with one single system and one person as main evaluator. However, all results were validated with a RiSE Event SPL developer, who participated of the whole process, from the test suite definition to the solutions for the problematic interactions. Nonetheless, the reader must be careful when generalizing results beyond the studied system.

7.7 CHAPTER SUMMARY

VarXplorer is an iterative and incremental approach designed to assist developers during the software development. From a test suite, the graphs generated by VarXplorer present how pairwise features interact with each other, through suppress and require relationships. The tool allows developers to define interactions as suspicious or benign through right clicks on the interaction (edge) that connects two features. Those interactions are used to remove benign interactions for the next test cases. This process reduces the amount of information that the developer needs to analyze towards finding problems in the system. Specifications clean the graphs to help developers to focus only on new interactions that are likely to present problems. The study presented in this chapter aimed to analyze how this iterative process reduces the complexity of identifying problems in graphs. We found that the specifications reduce by about 50% the amount of information that the user has to analyze. In the next chapter, we present the concluding remarks of our work and discuss potential future work.

PART V

CONCLUSIONS

Chapter 8

CONCLUDING REMARKS AND FUTURE WORK

Feature interactions occur when a feature behavior is influenced by the presence of another feature(s). Typically, interactions may lead to faults that are not easily identified from the analysis of each feature separately, specially when feature specifications are missing. Next, we present the contributions made by this thesis and directions to future work.

8.1 THESIS CONTRIBUTIONS

In this thesis, we are pursuing a twofold goal in the context of highly configurable systems: mapping the state-of-the-art on feature interactions and identifying suspicious interactions without upfront specifications. To fulfill our goals, we made the following contributions:

1. *A systematic mapping study (Chapter 3)*. We conducted a systematic mapping study with seven research questions, in which the 40 studies found are mainly classified regarding the feature interaction solution presented: detection, resolution and general analysis. More than 43% of the studies discussed how to identify interactions at early phases of the SPL development, mainly based on traceability, dependencies, verification of assertions, feature exclusion, precedence, and adaptation. Another 40% comprised approaches focused on source code to detect and resolve interactions. For example, they were based on: model checkers, non-functional properties measurements, conditional compilation, and derivatives. The remaining studies provided an initial discussion about feature interaction management, such as models, specification and ways to prevent interactions.

   In general, we observed the approaches are based on software specifications as the main strategy to detect interactions. In addition, instead of detecting interactions from the running software, they provide predictions based on models and static analysis of source code. Although they are able to detect interactions, deciding whether a given interaction is benign or represents a bug is still challenging. Furthermore, we noticed that the literature on feature interactions for SPL is poor in providing evaluations and case studies.

2. *VarXplorer (Chapters 4 and 5).* To overcome the drawbacks exemplified on previous item, we provide an approach to iteratively detect interactions and developed a tool (Eclipse plug-in). We aimed to contribute to the SPL community towards the identification of suspicious interactions. Our tool, VarXplorer, is able to identify interactions and relationships between features without upfront specifications.

   With VarXplorer, we provide an automatic way to identify feature interactions based on the software execution. From the execution of a test case, we analyze interactions based on its control and data flow. Moreover, we present additional indicators that help developers to identify which interactions may represent a bug, such as the suppression of one feature by another and the variables involved in the interaction.

3. *First Study: understanding interaction with the graph (Chapter 6).* VarXplorer proposes to organize interactions, conditional variables, and feature relationships as an interactive feature-interaction graph. To evaluate whether the graph do help developers to understand and identify suspicious interactions, we performed a controlled experiment with 24 subjects from two universities and four companies. We measured the effort of a participant, which had to identify a suspicious interaction using the information provided by the feature-interaction graph. Besides the statistical quantitative analysis, we performed an in-depth qualitative discussion based on video and audio recordings, and post-treatment interviews.

   The results confirmed that participants using VarXplorer are much faster compared to the state-of-the-art tool (Varviz). VarXplorer improves the performance of identifying suspicious interactions, and it is at least 3 times faster than Varviz. In addition, we also found that the relationships graphically represented as arrows and colors in VarXplorer make the developer work easier and faster. Also, VarXplorer only shows conditional variables, which reduces the amount of information shown to developers.

4. *Second Study: an analysis on iterations (Chapter 7).* In addition to provide feature-interaction graphs, VarXplorer also presents an iterative approach to identify interactions from a test suite, besides automatically documenting interactions through feature-interaction specifications. The first study focuses on understanding the graph and how fast an user identifies suspicious interactions compared to the state-of-the-art tool. Conversely, this study investigates the iterative process: how much effort we save using specifications, and how many interactions are cleaned from each graph during the testing process.

   The results showed that when using specifications, the developers see 45% less interactions on average and they have 50% less variables to analyze. In a test suite, where test cases are executed sequentially, one after the other, some tests may present similarities. Interactions judged as benign in previous tests are removed from the next graphs and the user sees a reduced graph, which has only new interactions specific to that test that have not been analyzed yet. VarXplorer aims to support members of a development team identifying real feature-interaction problems (at runtime) providing a friendly and graphical interface, and reduced information to make their work faster and more objective.

8.2 LIMITATIONS AND DIRECTIONS FOR FUTURE WORK

In this section, we provide an extra discussion on the directions for future work for VarXplorer. In addition, we also present potential gaps identified after carrying out the mapping study.

8.2.1 Potential Future Work for the Approach

Interaction detection. When using our approach, the user may spend less effort in finding problematic interactions. VarXplorer provides a visualization of all interactions in a configurable system and highlights feature relationships that may help users to find bugs. Although we cover all feature combinations in an execution, we use test cases to detect interactions, and, then, we may miss interactions present in uncovered inputs. So far, we are not aware of any testing approach focused on finding feature-interaction problems at runtime, which considers variational execution. For future work, we may propose a test-case generation to use in combination with our approach to cover the most representative inputs of a given system.

From the experimental studies, we identified a set of causes of interactions, such as, wrong object, misplaced variable overwrite, and incorrect conditional statements. We also found problematic interactions related to spread code and lack of modularity, which may led to unnecessary control and data dependencies. Although these last two problems may not necessarily cause software bugs, they negatively impact design quality and software evolution. For future work, we are interested in investigating how “interaction smells” impact on software quality; besides how we can find (testing approaches) and resolve them (recommendations).

VarXplorer uses the variability-aware interpreter, VarexJ, to generate variational traces [50]. Thus, our approach inherits VarexJ’s technical limitations. For example, it can only execute Java programs, and analyzing large systems may be computationally expensive. However, VarXplorer does not depend on VarexJ. The set of presence conditions collected during runtime can be obtained from other variability-aware execution approaches. Furthermore, we may also obtain information about feature interactions from symbolic execution [108], static analysis [107], or execution comparison [130].

Technical aspects. Our current approach focuses on pair-wise feature interactions. While higher-order interactions are less common in practice [50], they do still may lead to unexpected behavior. In the future, we aim to consider such interactions, which however come with challenges for scalability and appropriate visualization that need to be solved. In general, we plan to improve our current preliminary feature interaction graph to enable easier visualization of systems with a large set of features.

For systems that present many interactions and to overcome scalability issues, as future work we may allow developers to create sub-graphs. They will be able to choose which features they want to analyze, and then the tool can create a sub-graph containing only the interactions of the selected features.

Feature interaction specification language. The specification of interactions has two main benefits. Besides helping create the specifications of the system, it contributes to "clean" the graph by iteratively removing interactions that the user recognizes as desired or benign. In a graph with many interactions, we provide a way to incrementally remove benign interactions in each test case. Thus, users can focus their attention only on suspicious interactions that may represent a problem for the correct operation of the system. To make the first graph less cluttered and thus easier for the user to interpret, we could additionally consider already documented global and feature specifications to filter the interactions accordingly.

8.2.2 Other Directions for Future Work from the Systematic Mapping

Combining strategies. As a way to cover different aspects of an SPL, some strategies could be used together or even applied to different domains to maximize the use of the same approach before starting to define a new one. We found few studies that deal with effectiveness or suitability of combining strategies. For instance, Liu et al. [7] proposed a tool-based approach to support safe evolution of SPL requirements using a model-based approach. However, some discussions are still missing. For example, early detection approaches could benefit from structural (source code) and operational (data and control flows) analysis to evaluate the efficiency of previously detected interactions. The opposite could also be interesting, traceability from source code to models.

Domain of smartphones and apps. With the arrival of smartphones and applications, many other interactions problems may have emerged and could be further investigated, such as: (i) interactions among apps from the same supplier but different systems families; (ii) interactions among apps from different suppliers; (iii) interactions between an app and the mobile operating system; and also (iv) internal interactions to a single app, which is the most common interaction in the development of systems in general, and has been the focus of the research community that investigates feature interaction issues.

BIBLIOGRAPHY

1 PETERSEN, K.; FELDT, R.; MUJTABA, S.; MATTSSON, M. Systematic mapping studies in software engineering. In: *Proc. of the 12th Inter. Conference on Evaluation and Assessment in Software Engineering*. Swinton, UK: British Computer Society, 2008. (EASE), p. 68–77.

2 APEL, S.; SCHOLZ, W.; LENGAUER, C.; KÄSTNER, C. Detecting dependences and interactions in feature-oriented design. In: IEEE. *2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)*. [S.l.], 2010. p. 161–170.

3 APEL, S.; RHEIN, A. V.; THÜM, T.; KÄSTNER, C. Feature-interaction detection based on feature-based specifications. *Computer Networks*, Elsevier, v. 57, n. 12, p. 2399–2409, 2013.

4 APEL, S.; RHEIN, A. v.; WENDLER, P.; GRÖSSLINGER, A.; BEYER, D. Strategies for product-line verification: case studies and experiments. In: IEEE PRESS. *Proceedings of the 2013 International Conference on Software Engineering*. [S.l.], 2013. p. 482–491.

5 MUSSBACHER, G.; ARAÚJO, J.; MOREIRA, A.; AMYOT, D. AoURN-based modeling and analysis of software product lines. *Software Quality Journal*, Springer, v. 20, n. 3-4, p. 645–687, 2012.

6 SCHOLZ, W.; THÜM, T.; APEL, S.; LENGAUER, C. Automatic detection of feature interactions using the java modeling language: an experience report. In: ACM. *Proceedings of the 15th International Software Product Line Conference, Volume 2*. [S.l.], 2011. p. 7.

7 LIU, J.; DEHLINGER, J.; SUN, H.; LUTZ, R. State-based modeling to support the evolution and maintenance of safety-critical software product lines. In: *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*. [S.l.: s.n.], 2007. p. 596–608.

8 BESSLING, S.; HUHN, M. Towards formal safety analysis in feature-oriented product line development. In: SPRINGER. *International Symposium on Foundations of Health Informatics Engineering and Systems*. [S.l.], 2013. p. 217–235.

9 RAZZAQ, A.; ABBASI, R. Automated separation of crosscutting concerns: Earlier automated identification and modularization of cross-cutting features at analysis phase. In: IEEE. *2012 15th International Multitopic Conference (INMIC)*. [S.l.], 2012. p. 471–478.

10 LINSBAUER, L.; LOPEZ-HERREJON, R. E.; EGYED, A. Variability extraction and modeling for product variants. *Software & Systems Modeling*, Springer, p. 1–21, 2016.

11 BEN-DAVID, S.; STERIN, B.; ATLEE, J. M.; BEIDU, S. Symbolic model checking of product-line requirements using sat-based methods. In: IEEE. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*. [S.l.], 2015. v. 1, p. 189–199.

12 SCHUSTER, S.; SCHULZE, S.; SCHAEFER, I. Structural feature interaction patterns: Case studies and guidelines. In: *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*. New York, NY, USA: ACM, 2013. (VaMoS '14), p. 14:1–14:8. ISBN 978-1-4503-2556-1.

13 ATLEE, J. M.; FAHRENBERG, U.; LEGAY, A. Measuring behaviour interactions between product-line features. In: *Proceedings of the Third FME Workshop on Formal Methods in Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2015. (Formalise '15), p. 20–25.

14 BREDEREKE, J. Configuring members of a family of requirements using features. In: *Feature Interactions in Telecommunications and Software Systems VIII, ICFI'05, 28-30 June 2005, Leicester, UK*. [S.l.]: IOS Press, 2005. p. 96–113.

15 HU, H.; YANG, D.; FU, L.; XIANG, H.; FU, C.; SANG, J.; YE, C.; LI, R. Semantic web-based policy interaction detection method with rules in smart home for detecting interactions among user policies. *IET Communications*, v. 5, n. 17, p. 2451–2460, November 2011. ISSN 1751-8628.

16 METZGER, A.; BÜHNE, S.; LAUENROTH, K.; POHL, K. Considering feature interactions in product lines: Towards the automatic derivation of dependencies between product variants. In: *Feature Interactions in Telecommunications and Software Systems*. [S.l.: s.n.], 2005. p. 198–216.

17 CLASSEN, A.; HEYMANS, P.; SCHOBBENS, P.-Y. What's in a feature: A requirements engineering perspective. In: SPRINGER. *International Conference on Fundamental Approaches to Software Engineering*. [S.l.], 2008. p. 16–30.

18 ALFÉREZ, M.; MOREIRA, A.; KULESZA, U.; ARAÚJO, J.; MATEUS, R.; AMARAL, V. Detecting feature interactions in SPL requirements analysis models. In: ACM. *Proceedings of the First International Workshop on Feature-Oriented Software Development*. [S.l.], 2009. p. 117–123.

19 SHAKER, P.; ATLEE, J. M.; WANG, S. A feature-oriented requirements modelling language. In: IEEE. *Requirements Engineering Conference (RE), 2012 20th IEEE International*. [S.l.], 2012. p. 151–160.

20 BOCOVICH, C.; ATLEE, J. M. Variable-specific resolutions for feature interactions. In: ACM. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. [S.l.], 2014. p. 553–563.

21 KIM, C. H. P.; KÄSTNER, C.; BATORY, D. On the modularity of feature interactions. In: ACM. *Proceedings of the 7th international conference on Generative programming and component engineering.* [S.l.], 2008. p. 23–34.

22 BLUNDELL, C.; FISLER, K.; KRISHNAMURTHI, S.; HENTENRVCK, P. V. Parameterized interfaces for open system verification of product lines. In: *Proceedings. 19th International Conference on Automated Software Engineering, 2004.* [S.l.: s.n.], 2004. p. 258–267. ISSN 1938-4300.

23 LI, H. C.; KRISHNAMURTHI, S.; FISLER, K. Modular verification of open features using three-valued model checking. *Automated Software Engineering*, Springer, v. 12, n. 3, p. 349–382, 2005.

24 LIU, J.; BATORY, D.; LENGAUER, C. Feature oriented refactoring of legacy applications. In: ACM. *Proceedings of the 28th international conference on Software engineering.* [S.l.], 2006. p. 112–121.

25 SIEGMUND, N.; KOLESNIKOV, S. S.; KäSTNER, C.; APEL, S.; BATORY, D.; ROSENMüLLER, M.; SAAKE, G. Predicting performance via automated feature-interaction detection. In: *Proceedings of the 34th International Conference on Software Engineering.* Piscataway, NJ, USA: IEEE Press, 2012. (ICSE '12), p. 167–177. ISBN 978-1-4673-1067-3.

26 KÄSTNER, C.; APEL, S.; ROSENMÜLLER, M.; BATORY, D.; SAAKE, G. et al. On the impact of the optional feature problem: Analysis and case studies. In: CARNEGIE MELLON UNIVERSITY. *Proceedings of the 13th International Software Product Line Conference.* [S.l.], 2009. p. 181–190.

27 PADMANABHAN, P.; LUTZ, R. R. Tool-supported verification of product line requirements. *Automated Software Engineering*, Springer, v. 12, n. 4, p. 447–465, 2005.

28 SOCHOS, P.; RIEBISCH, M.; PHILIPPOW, I. The feature-architecture mapping (farm) method for feature-oriented development of software product lines. In: IEEE. *13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems - ECBS.* [S.l.], 2006. p. 9–pp.

29 BATORY, D.; HÖFNER, P.; MÖLLER, B.; ZELEND, A. Features, modularity, and variation points. In: ACM. *Proceedings of the 5th International Workshop on Feature-Oriented Software Development.* [S.l.], 2013. p. 9–16.

30 TAKEYAMA, F.; CHIBA, S. Implementing feature interactions with generic feature modules. In: SPRINGER. *International Conference on Software Composition.* [S.l.], 2013. p. 81–96.

31 PAREJO, J. A.; SáNCHEZ, A. B.; SEGURA, S.; RUIZ-CORTéS, A.; LOPEZ-HERREJON, R. E.; EGYED, A. Multi-objective test case prioritization in highly configurable systems: A case study. *Journal of Systems and Software*, v. 122, p. 287 – 310, 2016. ISSN 0164-1212.

32 LOTUFO, R.; SHE, S.; BERGER, T.; CZARNECKI, K.; WaSOWSKI, A. Evolution of the linux kernel variability model. In: . Berlin, Heidelberg: Springer-Verlag, 2010. (Proceedings of the International Software Product Lines Conference), p. 136–150. ISBN 3-642-15578-2, 978-3-642-15578-9.

33 ROTHBERG, V.; DINTZNER, N.; ZIEGLER, A.; LOHMANN, D. Feature models in linux: From symbols to semantics. In: . New York, NY, USA: ACM, 2016. (Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)), p. 65–72. ISBN 978-1-4503-4019-9.

34 APEL, S.; KOLESNIKOV, S.; SIEGMUND, N.; KäSTNER, C.; GARVIN, B. Exploring Feature Interactions in the Wild: The New Feature-interaction Challenge. In: *Proceedings of the 5th International Workshop on Feature-Oriented Software Development.* [S.l.: s.n.], 2013. (FOSD '13), p. 1–8. ISBN 978-1-4503-2168-6.

35 BOWEN, T. F.; DWORACK, F. S.; CHOW, C. H.; GRIFFETH, N.; HERMAN, G. E.; LIN, Y. J. The feature interaction problem in telecommunications systems. In: *Seventh International Conference on Software Engineering for Telecommunication Switching Systems (SETSS).* [S.l.: s.n.], 1989. p. 59–62.

36 COHEN, M. B.; DWYER, M. B.; SHI, J. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. *IEEE Transactions on Software Engineering*, v. 34, n. 5, p. 633–650, 2008.

37 THüM, T.; APEL, S.; KäSTNER, C.; SCHAEFER, I.; SAAKE, G. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 47, n. 1, p. 6:1–6:45, jun. 2014. ISSN 0360-0300.

38 KIM, C. H. P.; MARINOV, D.; KHURSHID, S.; BATORY, D.; SOUTO, S.; BARROS, P.; D&#039;AMORIM, M. Splat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering.* New York, NY, USA: ACM, 2013. (ESEC/FSE 2013), p. 257–267. ISBN 978-1-4503-2237-9.

39 SOUTO, S.; D'AMORIM, M.; GHEYI, R. Balancing Soundness and Efficiency for Practical Testing of Configurable Systems. In: IEEE PRESS. [S.l.], 2017. p. 632–642.

40 KIM, C. H. P.; BATORY, D.; KHURSHID, S. Eliminating products to test in a software product line. In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering.* New York, NY, USA: ACM, 2010. (ASE '10), p. 139–142. ISBN 978-1-4503-0116-9.

41 COHEN, M. B.; DWYER, M. B.; SHI, J. Interaction testing of highly-configurable systems in the presence of constraints. In: *Proceedings of the 2007 International Symposium on Software Testing and Analysis.* New York, NY, USA: ACM, 2007. (ISSTA '07), p. 129–139. ISBN 978-1-59593-734-6.

42 MEDEIROS, F.; KäSTNER, C.; RIBEIRO, M.; GHEYI, R.; APEL, S. A comparison of 10 sampling algorithms for configurable systems. In: *Proceedings of the 38th International Conference on Software Engineering*. New York, NY, USA: ACM, 2016. (ICSE '16), p. 643–654. ISBN 978-1-4503-3900-1.

43 NIE, C.; LEUNG, H. A survey of combinatorial testing. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 43, n. 2, p. 11:1–11:29, fev. 2011. ISSN 0360-0300.

44 LI, H. C.; KRISHNAMURTHI, S.; FISLER, K. Modular verification of open features using three-valued model checking. *Automated Software Engg.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 12, n. 3, p. 349–382, jul. 2005. ISSN 0928-8910.

45 APEL, S.; SPEIDEL, H.; WENDLER, P.; RHEIN, A. von; BEYER, D. Detection of feature interactions using feature-aware verification. In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2011. (ASE '11), p. 372–375. ISBN 978-1-4577-1638-6.

46 RHEIN, E. V.; APEL, S.; RAIMONDI, F. F.: Introducing binary decision diagrams in the explicit-state verification of java code. In: *In: Proc. Java Pathfinder Workshop*. [S.l.: s.n.], 2011.

47 CLASSEN, A.; HEYMANS, P.; SCHOBBENS, P.-Y.; LEGAY, A. Symbolic model checking of software product lines. In: *Proceedings of the 33rd International Conference on Software Engineering*. New York, NY, USA: ACM, 2011. (ICSE '11), p. 321–330. ISBN 978-1-4503-0445-0.

48 BURCH, J.; CLARKE, E.; MCMILLAN, K.; DILL, D.; HWANG, L. Symbolic Model Checking: 10^{20} States and Beyond. In: . [S.l.: s.n.], 1992. v. 98, n. 2, p. 142 – 170. ISSN 0890-5401.

49 NGUYEN, H. V.; KÄSTNER, C.; NGUYEN, T. N. Exploring variability-aware execution for testing plugin-based web applications. In: *Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: ACM, 2014. (ICSE 2014), p. 907–918. ISBN 978-1-4503-2756-5.

50 MEINICKE, J.; WONG, C. P.; KÄSTNER, C.; THÜM, T.; SAAKE, G. On essential configuration complexity: Measuring interactions in highly-configurable systems. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2016. (ASE 2016), p. 483–494. ISBN 978-1-4503-3845-5.

51 KIM, C. H. P.; KHURSHID, S.; BATORY, D. Shared execution for efficiently testing product lines. In: *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. [S.l.: s.n.], 2012. p. 221–230. ISSN 1071-9458.

52 AUSTIN, T. H.; FLANAGAN, C. Multiple Facets for Dynamic Information Flow. v. 47, n. 1, p. 165–178, 2012.

53 ATLEE, J. M.; FAHRENBERG, U.; LEGAY, A. Measuring behaviour interactions between product-line features. In: *Proceedings of the International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2015. p. 20–25.

54 MOSSER, S.; PARRA, C.; DUCHIEN, L.; BLAY-FORNARINO, M. Using domain features to handle feature interactions. In: *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*. New York, NY, USA: ACM, 2012. (VaMoS '12), p. 101–110. ISBN 978-1-4503-1058-1.

55 GIBSON, J. P.; LALLET, E.; RAFFY, J.-L. Feature interactions in a software product line for e-voting. In: NAKAMURA, M.; REIFF-MARGANIEC, S. (Ed.). *Feature Interactions in Telecommunications and Software Systems (ICFI)*. [S.l.]: IOS Press, 2009. p. 91–106. ISBN 978-1-60750-014-8.

56 BOWEN, T. F.; DWORACK, F. S.; CHOW, C. H.; GRIFFETH, N.; HERMAN, G. E.; LIN, Y. J. The feature interaction problem in telecommunications systems. In: *Software Engineering for Telecommunication Switching Systems, 1989. SETSS 89., Seventh International Conference on*. [S.l.: s.n.], 1989. p. 59–62.

57 SOARES, L. R.; SCHOBBENS, P.-Y.; MACHADO, I. do C.; ALMEIDA, E. S. de. Feature interaction in software product line engineering: A systematic mapping study. *Information and Software Technology*, p. –, 2018. ISSN 0950-5849.

58 SOARES, L. R.; MEINICKE, J.; NADI, S.; KäSTNER, C.; ALMEIDA, E. S. de. Varxplorer: Lightweight process for dynamic analysis of feature interactions. In: *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*. New York, NY, USA: ACM, 2018. (VAMOS 2018), p. 59–66. ISBN 978-1-4503-5398-4.

59 SOARES, L. R. Varxplorer: Reasoning about feature interactions. In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. New York, NY, USA: ACM, 2018. (ICSE '18), p. 500–502. ISBN 978-1-4503-5663-3.

60 SOARES, L. R.; MEINICKE, J.; NADI, S.; KäSTNER, C.; ALMEIDA, E. S. de. Exploring feature interactions without specifications: A controlled experiment. In: *Proceedings of the 17th International Conference on Generative Programming: Concepts Experience*. New York, NY, USA: ACM, 2018. (GPCE'18).

61 SOARES, L. R.; MACHADO, I. do C.; ALMEIDA, E. S. de. Non-functional properties in software product lines: A reuse approach. In: *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems*. New York, NY, USA: ACM, 2015. (VaMoS '15), p. 67:67–67:74. ISBN 978-1-4503-3273-6.

62 SOARES, L. R.; POTENA, P.; MACHADO, I. do C.; CRNKOVIC, I.; ALMEIDA, E. S. de. Analysis of non-functional properties in software product lines: A systematic review. In: *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*. [S.l.: s.n.], 2014. p. 328–335. ISSN 1089-6503.

63 VALE, T.; CABRAL, B.; ALVIM, L.; SOARES, L.; SANTOS, A.; MACHADO, I.; SOUZA, I.; FREITAS, I.; ALMEIDA, E. Splice: A lightweight software product line development process for small and medium size projects. In: IEEE COMPUTER SOCIETY. *Proceedings of the 2014 Eighth Brazilian Symposium on Software Components, Architectures and Reuse*. [S.l.], 2014. p. 42–52.

64 MEINICKE, J. *VarexJ: A Variability-Aware Interpreter for Java Applications*. Dissertação (Mestrado) — University of Magdeburg, dez. 2014.

65 Meinicke, J.; Wong, C.-P.; Kästner, C.; Saake, G. Understanding Differences among Executions with Variational Traces. *ArXiv e-prints*, jul. 2018.

66 CLEMENTS, P.; NORTHROP, L. *Software Product Lines: Practices and Patterns*. Boston, MA, USA: Addison-Wesley, 2001. ISBN 0201703327.

67 APEL, S.; KäSTNER, C. An overview of feature-oriented software development. *Journal of Object Technology*, v. 8, n. 5, p. 49–84, jul 2009. ISSN 1660-1769.

68 SOARES, L. R.; MACHADO, I. do C.; ALMEIDA, E. S. de. Non-functional properties in software product lines: A reuse approach. In: *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems*. New York, NY, USA: ACM, 2015. (VaMoS '15), p. 67:67–67:74. ISBN 978-1-4503-3273-6.

69 CZARNECKI, K.; EISENECKER, U. W. *Generative Programming: Methods, Tools, and Applications*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000. ISBN 0-201-30977-7.

70 APEL, S.; KÄSTNER, C. An overview of feature-oriented software development. *Journal of Object Technology*, v. 8, n. 5, p. 49–84, 2009.

71 KRUEGER, C. W. The biglever software gears unified software product line engineering framework. In: *2008 12th International Software Product Line Conference*. [S.l.: s.n.], 2008. p. 353–353.

72 BEUCHE, D. Modeling and building software product lines with pure:: variants. In: ACM. *Proceedings of the 16th International Software Product Line Conference-Volume 2*. [S.l.], 2012. p. 255–255.

73 THÜM, T.; KÄSTNER, C.; BENDUHN, F.; MEINICKE, J.; SAAKE, G.; LEICH, T. Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, Elsevier, v. 79, p. 70–85, 2014.

74 SCHMID, K.; EICHELBERGER, H. Easy-producer: From product lines to variability-rich software ecosystems. In: *Proceedings of the 19th International Conference on Software Product Line*. New York, NY, USA: ACM, 2015. (SPLC '15), p. 390–391. ISBN 978-1-4503-3613-0.

75 BATORY, D.; HöFNER, P.; KIM, J. Feature interactions, products, and composition. In: *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*. New York, NY, USA: ACM, 2011. (GPCE '11), p. 13–22. ISBN 978-1-4503-0689-8.

76 BRUNS, G. Foundations for features. In: *Feature Interactions in Telecommunications and Software Systems VIII, ICFI'05, 28-30 June 2005, Leicester, UK*. [S.l.: s.n.], 2005. p. 3–11.

77 MULTIMEDIA-LLC. Featureopt: Taming and optimizing feature interaction in software-intensive automotive systems. In: *Acessed in 07.23.2018*. [s.n.], 2018. Disponível em: ¡http://iktderzukunft.at/en/projects/feature-opt.php#contactAddress¿.

78 DOMINGUEZ, A. L. J. Feature interaction detection in the automotive domain. In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. [S.l.: s.n.], 2008. p. 521–524. ISSN 1938-4300.

79 ZAVE, P.; CHEUNG, E.; YAROSH, S. Toward user-centric feature composition for the internet of things. *ArXiv e-prints arXiv:1510.06714*, 2015.

80 CALDER, M.; KOLBERG, M.; MAGILL, E. H.; REIFF-MARGANIEC, S. Feature interaction: a critical review and considered forecast. *Computer Networks*, Elsevier North-Holland, Inc., v. 41, n. 1, p. 115–141, jan 2003. ISSN 1389-1286.

81 DIETRICH, D.; SHAKER, P.; ATLEE, J. M.; RAYSIDE, D.; GORZNY, J. Feature interaction analysis of the feature-oriented requirements-modelling language using Alloy. In: *Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation*. New York, NY, USA: ACM, 2012. (MoDeVVa '12), p. 17–22. ISBN 978-1-4503-1801-3.

82 RODRIGUES, I.; RIBEIRO, M.; MEDEIROS, F.; BORBA, P.; FONSECA, B.; GHEYI, R. Assessing fine-grained feature dependencies. *Inf. Softw. Technol.*, Butterworth-Heinemann, Newton, MA, USA, v. 78, n. C, p. 27–52, out. 2016. ISSN 0950-5849.

83 ABAL, I.; BRABRAND, C.; WASOWSKI, A. 42 variability bugs in the Linux kernel: A qualitative analysis. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2014. (ASE '14), p. 421–432. ISBN 978-1-4503-3013-8.

84 APEL, S.; KOLESNIKOV, S.; SIEGMUND, N.; KäSTNER, C.; GARVIN, B. Exploring feature interactions in the wild: The new feature-interaction challenge. In: *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*. New York, NY, USA: ACM, 2013. (FOSD '13), p. 1–8. ISBN 978-1-4503-2168-6.

85 HALL, R. J. Feature combination and interaction detection via foreground/background models. *Computer Networks*, v. 32, n. 4, p. 449 – 469, 2000. ISSN 1389-1286.

86 SCHOBBENS, P.-Y.; HEYMANS, P.; TRIGAUX, J.-C.; BONTEMPS, Y. Generic semantics of feature diagrams. *Comput. Netw.*, Elsevier North-Holland, Inc., New York, NY, USA, v. 51, n. 2, p. 456–479, fev. 2007. ISSN 1389-1286.

87 OHTA, T.; HARADA, Y. Classification, detection and resolution of service interactions in telecommunication services. *Feature Interactions in Telecommunications Systems*, IOS Press, p. 60, 1994.

88 PLATH, M.; RYAN, M. Feature integration using a feature construct. *Science of Computer Programming*, v. 41, n. 1, p. 53 – 84, 2001. ISSN 0167-6423.

89 PREHOFER, C. Feature-oriented programming: A fresh look at objects. In: *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*. [S.l.]: Springer, 1997. v. 1241, p. 419–443.

90 LIU, J. Feature interactions and software derivatives. *Journal of Object Technology*, v. 4, n. 3, p. 13–19, 2004.

91 NETO, P. A. da M. S.; MACHADO, I. do C.; MCGREGOR, J. D.; ALMEIDA, E. S. de; MEIRA, S. R. de L. A systematic mapping study of software product lines testing. *Inf. Softw. Technol.*, Butterworth-Heinemann, Newton, MA, USA, v. 53, n. 5, p. 407–423, maio 2011. ISSN 0950-5849.

92 BASTOS, J. F.; NETO, P. A. da M. S.; ALMEIDA, E. S. de; MEIRA, S. R. de L. Adopting software product lines: A systematic mapping study. In: *Evaluation Assessment in Software Engineering (EASE 2011), 15th Annual Conference on*. [S.l.: s.n.], 2011. p. 11–20.

93 SILVA, I. F. da; NETO, P. A. da M. S.; O'LEARY, P.; ALMEIDA, E. S. de; MEIRA, S. R. de L. Agile software product lines: A systematic mapping study. *Softw. Pract. Exper.*, John Wiley & Sons, Inc., New York, NY, USA, v. 41, n. 8, p. 899–920, jul. 2011. ISSN 0038-0644.

94 VALE, T.; ALMEIDA, E. S. de; ALVES, V.; KULESZA, U.; NIU, N.; LIMA, R. de. Software product lines traceability: A systematic mapping study. *Information and Software Technology*, v. 84, p. 1 – 18, 2017. ISSN 0950-5849.

95 KITCHENHAM, B. A.; BUDGEN, D.; BRERETON, P. (Ed.). *Evidence-Based Software Engineering and Systematic Reviews*. Boca Raton, FL, USA: Chapman and Hall/CRC, 2015. ISBN 1482228653.

96 ABDESSALEM, R. B.; PANICHELLA, A.; NEJATI, S.; BRIAND, L. C.; STIFTER, T. Testing autonomous cars for feature interaction failures using many-objective search. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2018. (ASE 2018), p. 143–154. ISBN 978-1-4503-5937-5.

97 FILHO, R. S. S.; REDMILES, D. F. Managing feature interaction by documenting and enforcing dependencies in software product lines. *Feature Interactions in Software and Communication Systems IX*, IOS Press, v. 33, 2008.

98 JAYARAMAN, P.; WHITTLE, J.; ELKHODARY, A. M.; GOMAA, H. Model composition in product lines and feature interaction detection using critical pair analysis. In: SPRINGER. *International Conference on Model Driven Engineering Languages and Systems*. [S.l.], 2007. p. 151–165.

99 ZHANG, Y.; GUO, J.; BLAIS, E.; CZARNECKI, K.; YU, H. A mathematical model of performance-relevant feature interactions. In: *Proceedings of the 20th International Systems and Software Product Line Conference*. New York, NY, USA: ACM, 2016. (SPLC '16), p. 25–34. ISBN 978-1-4503-4050-2.

100 NGUYEN, T.; KOC, U.; CHENG, J.; FOSTER, J. S.; PORTER, A. A. igen: Dynamic interaction inference for configurable software. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2016. (FSE 2016), p. 655–665. ISBN 978-1-4503-4218-6.

101 ZIBAEENEJAD, M. H.; ZHANG, C.; ATLEE, J. M. Continuous variable-specific resolutions of feature interactions. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: ACM, 2017. (ESEC/FSE 2017), p. 408–418. ISBN 978-1-4503-5105-8.

102 HU, H.; YANG, D.; FU, L.; XIANG, H.; FU, C.; SANG, J.; YE, C.; LI, R. Semantic web-based policy interaction detection method with rules in smart home for detecting interactions among user policies. *IET communications*, IET, v. 5, n. 17, p. 2451–2460, 2011.

103 CALDER, M.; KOLBERG, M.; MAGILL, E. H.; REIFF-MARGANIEC, S. Feature interaction: a critical review and considered forecast. *Computer Networks*, v. 41, n. 1, p. 115 – 141, 2003. ISSN 1389-1286.

104 RHEIN, A. von; GREBHAHN, A.; APEL, S.; SIEGMUND, N.; BEYER, D.; BERGER, T. Presence-condition simplification in highly configurable systems. In: . Piscataway, NJ, USA: IEEE Press, 2015. (ICSE), p. 178–188. ISBN 978-1-4799-1934-5.

105 ANGERER, F.; GRIMMER, A.; PRÄHOFER, H.; GRÜNBACHER, P. Configuration-Aware Change Impact Analysis. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.: s.n.], 2016. v. 00, p. 385–395.

106 BöHME, M.; OLIVEIRA, B. C. d. S.; ROYCHOUDHURY, A. Regression tests to expose change interaction errors. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: ACM, 2013. (ESEC/FSE 2013), p. 334–344. ISBN 978-1-4503-2237-9.

107 LILLACK, M.; KÄSTNER, C.; BODDEN, E. Tracking load-time configuration options. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2014. (ASE '14), p. 445–456. ISBN 978-1-4503-3013-8.

108 REISNER, E.; SONG, C.; MA, K.-K.; FOSTER, J. S.; PORTER, A. Using symbolic evaluation to understand behavior in configurable software systems. In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. New York, NY, USA: ACM, 2010. (ICSE '10), p. 445–454. ISBN 978-1-60558-719-6.

109 LAUENROTH, K.; POHL, K.; TOEHNING, S. Model checking of domain artifacts in product line engineering. In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009. (ASE '09), p. 269–280. ISBN 978-0-7695-3891-4.

110 KäSTNER, C.; RHEIN, A. von; ERDWEG, S.; PUSCH, J.; APEL, S.; RENDEL, T.; OSTERMANN, K. Toward variability-aware testing. In: *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*. New York, NY, USA: ACM, 2012. (FOSD '12), p. 1–8. ISBN 978-1-4503-1309-4.

111 NADI, S.; BERGER, T.; KäSTNER, C.; CZARNECKI, K. Mining configuration constraints: Static analyses and empirical results. In: . New York, NY, USA: ACM, 2014. (ICSE), p. 140–151. ISBN 978-1-4503-2756-5.

112 MAITY, S.; NAYAK, A. Improved test generation algorithms for pair-wise testing. In: *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*. [S.l.: s.n.], 2005. p. 10 pp.–244. ISSN 1071-9458.

113 AVILA-GEORGE, H.; TORRES-JIMENEZ, J.; IZQUIERDO-MARQUEZ, I. Improved pairwise test suites for non-prime-power orders. *IET Software*, v. 12, n. 3, p. 215–224, 2018. ISSN 1751-8806.

114 LIEBIG, J.; RHEIN, A. von; KÄSTNER, C.; APEL, S.; DÖRRE, J.; LENGAUER, C. Scalable analysis of variable software. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: ACM, 2013. (ESEC/FSE 2013), p. 81–91. ISBN 978-1-4503-2237-9.

115 HEIMAN, G. W. *Basic statistics for the behavioral sciences*. [S.l.]: Cengage Learning, 2013.

116 DEAN, A. Experimental design: Overview. In: SMELSER, N. J.; BALTES, P. B. (Ed.). *International Encyclopedia of the Social Behavioral Sciences*. Oxford: Pergamon, 2001. p. 5090 – 5096. ISBN 978-0-08-043076-8.

117 BEYER, H.; HOLTZBLATT, K. *Contextual Design: Defining Customer-Centered Systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN 1-55680-411-1.

118 BEN-DAVID, S.; STERIN, B.; ATLEE, J. M.; BEIDU, S. Symbolic model checking of product-line requirements using sat-based methods. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. Piscataway, NJ, USA: IEEE Press, 2015. (ICSE '15), p. 189–199. ISBN 978-1-4799-1934-5.

119 APEL, S.; RHEIN, A. v.; WENDLER, P.; LINGER, A. G.; BEYER, D. Strategies for product-line verification: Case studies and experiments. In: *Proceedings of the 2013 International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 482–491. ISBN 978-1-4673-3076-3.

120 APEL, S.; RHEIN, A. V.; THüM, T.; KäSTNER, C. Feature-interaction detection based on feature-based specifications. *Comput. Netw.*, Elsevier North-Holland, Inc., New York, NY, USA, v. 57, n. 12, p. 2399–2409, ago. 2013. ISSN 1389-1286.

121 GRIFFETH, N.; BLUMENTHAL, R.; GREGOIRE, J.-C.; OHTA, T. Feature interaction detection contest of the fifth international workshop on feature interactions. *Computer Networks*, v. 32, n. 4, p. 487 – 510, 2000. ISSN 1389-1286.

122 BOX, G. E.; HUNTER, J. S.; HUNTER, W. G. *Statistics for experimenters: design, innovation, and discovery*. [S.l.]: Wiley-Interscience New York, 2005. v. 2.

123 MELO, J.; BRABRAND, C.; WaSOWSKI, A. How Does the Degree of Variability Affect Bug Finding? In: . [S.l.]: ACM, 2016. p. 679–690.

124 LILLACK, M.; KÄSTNER, C.; BODDEN, E. Tracking Load-Time Configuration Options. IEEE, 2017.

125 ARZT, S.; RASTHOFER, S.; FRITZ, C.; BODDEN, E.; BARTEL, A.; KLEIN, J.; TRAON, Y. L.; OCTEAU, D.; MCDANIEL, P. Flowdroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. ACM, v. 49, n. 6, p. 259–269, 2014.

126 WONG, C.-P.; MEINICKE, J.; LAZAREK, L.; KÄSTNER, C. Faster Variational Execution with Transparent Bytecode Transformation. In: ACM. [S.l.], 2018.

127 HENTSCHEL, M.; HÄHNLE, R.; BUBEL, R. The interactive verification debugger: Effective understanding of interactive proof attempts. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2016. (ASE 2016), p. 846–851. ISBN 978-1-4503-3845-5.

128 ZELLER, A. Isolating cause-effect chains from computer programs. In: *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2002. (SIGSOFT '02/FSE-10), p. 1–10. ISBN 1-58113-514-9.

129 SUMNER, W. N.; ZHANG, X. Algorithms for automatically computing the causal paths of failures. In: *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held As Part of the Joint European Conferences*

on Theory and Practice of Software, ETAPS 2009. Berlin, Heidelberg: Springer-Verlag, 2009. (FASE '09), p. 355–369. ISBN 978-3-642-00592-3.

130 SUMNER, W. N.; ZHANG, X. Comparative causality: Explaining the differences between executions. In: *Proceedings of the 2013 International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 272–281.

131 APEL, S.; SPEIDEL, H.; WENDLER, P.; RHEIN, A. von; BEYER, D. Detection of Feature Interactions Using Feature-Aware Verification. In: . [S.l.: s.n.], 2011. p. 372–375.

132 NETO, P. A. d. M. S.; SANTANA, T. L. d.; ALMEIDA, E. S. d.; CAVALCANTI, Y. C. Rise events — a testbed for software product lines experimentation. In: *2016 IEEE/ACM 1st International Workshop on Variability and Complexity in Software Design (VACE)*. [S.l.: s.n.], 2016. p. 12–13.

133 AMMANN, P.; OFFUTT, J. *Introduction to Software Testing*. 1. ed. New York, NY, USA: Cambridge University Press, 2008. ISBN 0521880386, 9780521880381.

Appendix

A

SYSTEMATIC MAPPING: SUPPORT MATERIAL

This appendix presents extra information related to the mapping study presented in this thesis, earlier addressed in Chapter 3. In the next sections, we show the summary of primary studies discussed in the mapping.

A.1 SUMMARY OF STUDIES

Table A.1: Primary studies

Study	**Title**	**Reference**	**Category**	**Lifecycle**	**FI type**	**Tool**
P1	Parameterized interfaces for open system verification of product lines	[22]	De	DD	functional	
P2	Considering feature interactions in product lines: Towards the automatic derivation of dependencies between product variants	[16]	De	DA	functional	
P3	Modular verification of open features using three-valued model checking	[23]	De	DD	functional	
P4	Tool-supported verification of product line requirements	[27]	Re	DA/DD	functional	✓
P5	Configuring members of a family of requirements using features	[14]	Ae	DA	functional	✓

Table A.1: Primary studies

Study	Title	Reference	Category	Lifecycle	FI type	Tool
P6	The feature-architecture mapping (FArM) method for feature-oriented development of software product lines	[28]	Re	DA/DD	functional	✓
P7	Feature oriented refactoring of legacy applications	[24]	Rsc	DI/PC	structural	✓
P8	Managing feature interaction by documenting and enforcing dependencies in software product lines	[97]	Rsc	DA/DD/DI	structural	
P9	State-based modeling to support the evolution and maintenance of safety-critical software product lines	[7]	De	DA/DD	functional	✓
P10	Model composition in product lines and feature interaction detection using critical pair analysis	[98]	De/Re	DD	functional	✓
P11	On the modularity of feature interactions	[21]	Asc	DI/PC	structural	✓
P12	What's in a feature: A requirements engineering perspective	[17]	De	DA/DD	functional	✓
P13	Feature interactions in a software product line for E-voting	[55]	Ae	DD	functional	
P14	Detecting feature interactions in SPL requirements analysis models	[18]	De	DA/DD	functional	
P15	On the impact of the optional feature problem: Analysis and case studies	[26]	Rsc	DI/PC	structural	
P16	Detecting dependences and interactions in feature-oriented design	[2]	De/Re	DA/DD	functional	✓
P17	Semantic web-based policy interaction detection method with rules in smart home for detecting interactions among user policies	[102]	De	DA/DD	functional	
P18	Automatic detection of feature interactions using the java modeling language: An experience report	[6]	Dsc	DD/DI/PC	structural	✓
P19	Automated separation of crosscutting concerns: Earlier automated identification and modularization of crosscutting features at analysis phase	[9]	De	DA/DD	functional	✓
P20	AoURN-based modeling and analysis of software product lines	[5]	De/Re	DA/DD	functional	✓
P21	Using domain features to handle feature interactions	[54]	Re	DA/DD/PC	functional	
P22	Feature interaction analysis of the feature-oriented requirements-modelling language using alloy	[81]	De	DD	functional	✓

Table A.1: Primary studies

Study	Title	Reference	Category	Lifecycle	FI type	Tool
P23	A feature-oriented requirements modelling language	[19]	Ae	DA/DD	functional/ intended	
P24	Predicting performance via automated feature-interaction detection	[25]	Dsc	PC	non-func	✓
P25	Features, modularity, and variation points	[29]	Rsc	DD/DI/PC	structural	
P26	Feature-interaction detection based on feature-based specifications	[3]	Dsc	DD/DI/PC	functional	✓
P27	Implementing feature interactions with generic feature modules	[30]	Rsc	DI/PC	structural/ intended	
P28	Towards formal safety analysis in feature-oriented product line development	[8]	De	DA/DD	functional	✓
P29	Strategies for product-line verification: case studies and experiments	[4]	DSC	DI/PC	functional	✓
P30	Variable-specific resolutions for feature interactions	[20]	Re	DA/DD	functional	
P31	Structural feature interaction patterns: case studies and guidelines	[12]	Dsc	DI	structural	
P32	Symbolic model checking of product-line requirements using SAT-based methods	[11]	De	DA/DD	functional	✓
P33	Measuring behaviour interactions between product-line features	[13]	De	DD	functional	
P34	Variability extraction and modeling for product variants	[10]	Dsc/Rsc	DI/PC	structural	
P35	Assessing fine-grained feature dependencies	[82]	Dsc	DI	operational	✓

A.2 LIST OF THE NEWEST STUDIES, COLLECTED AFTER THE UPDATE

Table A.2: Continuation of the Primary studies

Study	Title	Reference	Category	Lifecycle	FI type	Tool
P36	A Mathematical Model of Performance-Relevant Feature Interactions	[22]	Dsc	PC	non-func	
P37	Continuous Variable-Specific Resolutions of Feature Interactions	[16]	Rsc/runtime	DI/PC	functional	
P38	iGen: Dynamic Interaction Inference for Configurable Software	[23]	Dsc/runtime	PC	functional	✓
P39	Testing Autonomous Cars for Feature Interaction Failures using Many-Objective Search	[82]	De	DD	functional	
P40	On Essential Configuration Complexity: Measuring Interactions in Highly-Configurable Systems	[82]	Dsc/runtime	PC	functional	✓

Appendix

B

CONTROLLED EXPERIMENT: DATA AND SUPPORT MATERIAL

This appendix lists the documentation, data, script, and support material used in the experimental study, earlier addressed in Chapter 6.

This appendix is organized as follows:

Section B.1 presents the online survey used to get the profile of each participant;

Section B.2 shows the experiment tasks, i.e., the instructions followed by the participants during the experiment execution;

Section B.3 presents the time spent by each participant to execute the tasks;

Section B.4 shows the R script used for the statistical analysis;

B.1 ONLINE PRE-SURVEY (BACKGROUND FORM)

Background Form

Controlled Experiment Background Form

* Required

1. **Email address** *

2. **Full Name** *

3. **Institution** *

4. **Institution position (please mark all options that apply to you)** *
   Check all that apply.

   ☐ Undergraduate student
   ☐ Master Student
   ☐ PhD. student
   ☐ Tester
   ☐ Software engineer
   ☐ Developer
   ☐ Other: ____________________

TECHNICAL KNOWLEDGE

5. **How many years of programming experience do you have using any programming language?** *
   Mark only one oval.

   ◯ < 1 year
   ◯ >= 1 year and < 5 years
   ◯ >= 5 years and < 10 years
   ◯ >= 10 years

6. **How many years of experience do you have in JAVA development?** *
   Mark only one oval.

   - ⬭ < 1 year
   - ⬭ >= 1 year and < 5 years
   - ⬭ >= 5 years and < 10 years
   - ⬭ >= 10 years

7. **How many years of experience do you have in using Eclipse IDE for programming?** *
   Mark only one oval.

   - ⬭ < 1 year
   - ⬭ >= 1 year and < 5 years
   - ⬭ >= 5 years and < 10 years
   - ⬭ >= 10 years

Skip to question 7.

Additional Information

8. **Regarding your Highly Configurable Systems (HCS) and Software Product Line (SPL) background knowledge** *
   Mark only one oval.

   - ⬭ I have been involved in software development teams applying HCS/SPL techniques
   - ⬭ I am a researcher working on topics related to HCS/SPL Development
   - ⬭ I know what HCS/SPL are but I have never participated in a software project applying HCS/SPL development
   - ⬭ I have never heard about HCS/SPL
   - ⬭ Other: ______________________

B.2 EVALUATION: INSTRUCTIONS FOR PARTICIPANTS

Group 1

Schedule

1. Warm-up 1: Varviz - WordPress
2. Task 1: VarXplorer - Telephone
3. Warm-up 2: VarXplorer - Wordpress
4. Task 2: Varviz - Elevator

Warm-up: Varviz - WordPress

Steps:

- **Feature interaction definition**
- **Experiment objective: identifying bugs related to feature interactions**
- **Introduction to Varviz: trace (diamonds, expressions, statements, variables)**
- **Example based on the WordPress**

WordPress main Features:

· **Smiley**: replaces characters with smiley faces.

· **Fahrenheit**: displays temperature in Fahrenheit

· **Weather**: presents the current weather in either Celsius or Fahrenheit

Test case scenario:

· you are given a WordPress test case that shows the current temperature to the user.

Task Description

1) Try to understand how the features interact. In this case, **you should leverage the Varviz plug-in** when executing the test case.
2) Describe what you understand from the interactions presented in the graph.
3) Is there any suspicious interaction? Something that does not agree with the requirements? Why?

Task #1 Varviz - ELEVATOR

You have at most 30 minutes to try understanding the interactions and tell us whether they represent a bug or not.

Elevator: It is an elevator model that is extensible by various features such as stopping if the elevator is empty or priority service for a special floor.

Elevator main Features:

- **2/3- full**: When the lift detects that it is more than two-thirds full, it does not stop in response to landing calls, since it is unlikely to be able to accept more passengers. Instead, it gives priority to passengers already inside the lift, as serving them will help reduce its load.
- **Overloaded**: When the lift is overloaded, the doors will not close. Some passengers must get out.
- **Executive floor**: The lift gives priority to calls from the executive floor.
- **Weight**: Updates the weight when someone gets inside or gets out.

Program details:

- ***Timeshift*** method: it executes one movement per time: either moves up or moves down or opens the door (someone gets in) or closes the door (and moves)
- ***respectFloorCalls*** variable (related to the feature **2/3- full)**:
    - Boolean value that when false the elevator does not attend the calls until the weight is less than 2/3.
- Additional information:
    - **Max Weight: 100kg**
    - **Executive floor: 4th**
    - **2/3 full: above 67kg**

Test case scenario:

Three people need to use the elevator. The first two (Alice and Angeline) call at the same time when the elevator is idle at the 4th floor. Then, after it picks the 2nd girl and right after closing the door at the 2nd floor, the 3rd person (Bob) calls it. The elevator, then, moves two times (either goes up or goes down). What does it should do in those movements?

- Elevator starts on 4th floor
- 2 people call at the same time:
    - Call: Alice (40kg) from 3rd to 0 floor
    - Call: Angelina (40kg) from 2nd to 1st floor
- Elevator moves until it picks Angelina up.
- New call when the elevator is in the 2nd floor: Bob from 4th to 0 floor
- Elevator moves 2 times

Task Description

4) Try to understand how the features interact. In this case, **you should leverage the VarXplorer plug-in** when executing the tesrt.
5) Describe what you understand from the interactions presented in the graph.
6) Are there any suspicious interaction? Why?

Warm-up: VarXplorer - WordPress

Steps:

- **Introduction to VarXplorer: features and relationships (suppress and enable)**
- **Example based on the WordPress**

WordPress main Features:

- **Smiley**: replaces characters to smile faces.
- **Fahrenheit**: temperature is displayed in Fahrenheit
- **Weather**: presents the current weather in either Celsius or Fahrenheit

Test case scenario:

- The program shows the current temperature to the user.

Task Description

1) Try to understand how the features interact. In this case, **you should leverage the VarXplorer plug-in** when execute the tasks.
2) Describe what you understand from the interactions presented in the graph.
3) Is there any suspicious interaction? Why?

Task #1 TELEPHONE SYSTEM

You have at most 30 minutes to try understanding the interactions and tell us whether they represent a bug or not.

Telephone main Features:

- **Call forwarding busy line (CFB):** All calls to the subscribing line are redirected to a predetermined number when the line is busy.
- **Call Waiting**: allows the subscriber to be noticed that another party (incoming call) is trying to reach his number while her line is busy, and to accept the new call by placing the original call on hold.

Program details:

- ***Timeshift()*** method**:** executes one action per time (eg., setting caller as off/on hook, starting ringing)
- ***disconnectCaller()*** method**:** disconnects the caller (putting as on hook)
- ***Call.onhold*** variable**:** true when call is on waiting (controlled by the feature **Call Waiting**)

Test case scenario:

The system receives a call from Alice to Sophie. Unfortunately, Sophie is busy and cannot receive the call. Sophie has enabled the *call waiting* option to be executed when she is busy.

- Alice calls to Sophie
- Sophie is busy
- Sophie is subscribed to call waiting.
- Sophie did not choose any person to the call be forwarded.

Task Description

7) Try to understand how the features interact. In this case, **you should leverage the Varviz plug-in** when execute the tasks.
8) Describe what you understand from the interactions you can perceive in the trace.
9) Is there any suspicious interaction? Why?

Group 2

Schedule

5. Warm-up 1: Varviz - WordPress
6. Task 1: VarXplorer - Telephone
7. Warm-up 2: VarXplorer - Wordpress
8. Task 2: Varviz - Elevator

Warm-up: Varviz - WordPress

Steps:

- **Feature interaction definition**
- **Experiment objective: identifying bugs related to feature interactions**
- **Introduction to Varviz: trace (diamonds, expressions, statements, variables)**
- **Example based on the WordPress**

WordPress main Features:

· **Smiley**: replaces characters with smiley faces.
· **Fahrenheit**: displays temperature in Fahrenheit
· **Weather**: presents the current weather in either Celsius or Fahrenheit

Test case scenario:

· you are given a WordPress test case that shows the current temperature to the user.

Task Description

1) Try to understand how the features interact. In this case, **you should leverage the Varviz plug-in** when executing the test case.
2) Describe what you understand from the interactions presented in the graph.
3) Is there any suspicious interaction? Something that does not agree with the requirements? Why?

Task #1 TELEPHONE SYSTEM

You have at most 30 minutes to try understanding the interactions and tell us whether they represent a bug or not.

Telephone main Features:

- **Call forwarding busy line (CFB):** All calls to the subscribing line are redirected to a predetermined number when the line is busy.
- **Call Waiting**: allows the subscriber to be noticed that another party (incoming call) is trying to reach his number while her line is busy, and to accept the new call by placing the original call on hold.

Program details:

- ***Timeshift()*** method**:** executes one action per time (eg., setting caller as off/on hook, starting ringing)
- ***disconnectCaller()*** method**:** disconnects the caller (putting as on hook)
- ***Call.onhold*** variable**:** true when call is on waiting (controlled by the feature **Call Waiting**)

Test case scenario:

The system receives a call from Alice to Sophie. Unfortunately, Sophie is busy and cannot receive the call. Sophie has enabled the *call waiting* option to be executed when she is busy.

- Alice calls to Sophie
- Sophie is busy
- Sophie is subscribed to call waiting.
- Sophie did not choose any person to the call be forwarded.

Task Description

7) Try to understand how the features interact. In this case, **you should leverage the Varviz plug-in** when execute the tasks.
8) Describe what you understand from the interactions you can perceive in the trace.
9) Is there any suspicious interaction? Why?

Warm-up: VarXplorer - WordPress

Steps:

- **Introduction to VarXplorer: features and relationships (suppress and enable)**
- **Example based on the WordPress**

WordPress main Features:

- **Smiley**: replaces characters to smile faces.
- **Fahrenheit**: temperature is displayed in Fahrenheit
- **Weather**: presents the current weather in either Celsius or Fahrenheit

Test case scenario:

- The program shows the current temperature to the user.

Task Description

1) Try to understand how the features interact. In this case, **you should leverage the VarXplorer plug-in** when execute the tasks.
2) Describe what you understand from the interactions presented in the graph.
3) Is there any suspicious interaction? Why?

Task #1 VarXplorer - ELEVATOR

You have at most 30 minutes to try understanding the interactions and tell us whether they represent a bug or not.

Elevator: It is an elevator model that is extensible by various features such as stopping if the elevator is empty or priority service for a special floor.

Elevator main Features:

- **2/3- full**: When the lift detects that it is more than two-thirds full, it does not stop in response to landing calls, since it is unlikely to be able to accept more passengers. Instead, it gives priority to passengers already inside the lift, as serving them will help reduce its load.
- **Overloaded**: When the lift is overloaded, the doors will not close. Some passengers must get out.
- **Executive floor**: The lift gives priority to calls from the executive floor.
- **Weight**: Updates the weight when someone gets inside or gets out.

Program details:

- ***Timeshift*** method: it executes one movement per time: either moves up or moves down or opens the door (someone gets in) or closes the door (and moves)
- ***respectFloorCalls*** variable (related to the feature **2/3- full)**:
    - Boolean value that when false the elevator does not attend the calls until the weight is less than 2/3.
- Additional information:
    - **Max Weight: 100kg**
    - **Executive floor: 4th**
    - **2/3 full: above 67kg**

Test case scenario:

Three people need to use the elevator. The first two (Alice and Angeline) call at the same time when the elevator is idle at the 4th floor. Then, after it picks the 2nd girl up and right after closing the door at the 2nd floor, the 3rd person (Bob) calls it. The elevator, then, moves two times (either goes up or goes down). What does it should do in those movements?

- Elevator starts on 4th floor
- 2 people call at the same time:
    - Call: Alice (40kg) from 3rd to 0 floor
    - Call: Angelina (40kg) from 2nd to 1st floor
- Elevator moves until it picks Angelina up.
- New call when the elevator is in the 2nd floor: Bob from 4th to 0 floor
- Elevator moves 2 times

Task Description

4) Try to understand how the features interact. In this case, **you should leverage the VarXplorer plug-in** when executing the tesrt.
5) Describe what you understand from the interactions presented in the graph.
6) Are there any suspicious interaction? Why?

Group 3

Schedule

9. Warm-up 1: VarXplorer - WordPress
10. Task 1: VarXplorer - Telephone
11. Warm-up 2: Varviz - Wordpress
12. Task 2: Varviz - Elevator

Warm-up: VarXplorer - WordPress

Steps:

- **Feature interaction definition**
- **Experiment objective: identifying bugs related to feature interactions**
- **Introduction to VarXplorer: trace (diamonds, expressions, statements, variables)**
- **Example based on the WordPress**

WordPress main Features:

· **Smiley**: replaces characters with smiley faces.
· **Fahrenheit**: displays temperature in Fahrenheit
· **Weather**: presents the current weather in either Celsius or Fahrenheit

Test case scenario:

· you are given a WordPress test case that shows the current temperature to the user.

Task Description

1) Try to understand how the features interact. In this case, **you should leverage the Varviz plug-in** when executing the test case.
2) Describe what you understand from the interactions presented in the graph.
3) Is there any suspicious interaction? Something that does not agree with the requirements? Why?

Task #1 TELEPHONE SYSTEM

You have at most 30 minutes to try understanding the interactions and tell us whether they represent a bug or not.

Telephone main Features:

- **Call forwarding busy line (CFB):** All calls to the subscriber line are redirected to a predetermined number when the line is busy.
- **Call Waiting**: allows the subscriber to be noticed that another party (incoming call) is trying to reach his number while her line is busy, and to accept the new call by placing the original call on hold.

Program details:

- ***Timeshift()*** method**:** executes one action per time (eg., setting caller as off/on hook, starting ringing)
- ***disconnectCaller()*** method**:** disconnects the caller (putting as on hook)
- ***Call.onhold*** variable**:** true when call is on waiting (controlled by the feature **Call Waiting**)

Test case scenario:

The system receives a call from Alice to Sophie. Unfortunately, Sophie is busy and cannot receive the call. Sophie has enabled the *call waiting* option to be executed when she is busy.

- Alice calls to Sophie
- Sophie is busy
- Sophie is subscribed to call waiting.
- Sophie did not choose any person to the call be forwarded.

Task Description

7) Try to understand how the features interact. In this case, **you should leverage the VarXplorer plug-in** when execute the tasks.
8) Describe what you understand from the interactions you can perceive in the trace.
9) Is there any suspicious interaction? Why?

Warm-up: Varviz - WordPress

Steps:

- **Introduction to VarXplorer: features and relationships (suppress and enable)**
- **Example based on the WordPress**

WordPress main Features:

- **Smiley**: replaces characters to smile faces.
- **Fahrenheit**: temperature is displayed in Fahrenheit
- **Weather**: presents the current weather in either Celsius or Fahrenheit

Test case scenario:

- The program shows the current temperature to the user.

Task Description

1) Try to understand how the features interact. In this case, **you should leverage the Varviz plug-in** when execute the tasks.
2) Describe what you understand from the interactions presented in the graph.
3) Is there any suspicious interaction? Why?

Task #1 Varviz - ELEVATOR

You have at most 30 minutes to try understanding the interactions and tell us whether they represent a bug or not.

Elevator: It is an elevator model that is extensible by various features such as stopping if the elevator is empty or priority service for a special floor.

Elevator main Features:

- **2/3- full**: When the lift detects that it is more than two-thirds full, it does not stop in response to landing calls, since it is unlikely to be able to accept more passengers. Instead, it gives priority to passengers already inside the lift, as serving them will help reduce its load.
- **Overloaded**: When the lift is overloaded, the doors will not close. Some passengers must get out.
- **Executive floor**: The lift gives priority to calls from the executive floor.
- **Weight**: Updates the weight when someone gets inside or gets out.

Program details:

- ***Timeshift*** method: it executes one movement per time: either moves up or moves down or opens the door (someone gets in) or closes the door (and moves)
- ***respectFloorCalls*** variable (related to the feature **2/3- full)**:
    - Boolean value that when false the elevator does not attend the calls until the weight is less than 2/3.
- Additional information:
    - **Max Weight: 100kg**
    - **Executive floor: 4th**
    - **2/3 full: above 67kg**

Test case scenario:

Three people need to use the elevator. The first two (Alice and Angeline) call at the same time when the elevator is idle at the 4th floor. Then, after it picks the 2nd girl up and right after closing the door at the 2nd floor, the 3rd person (Bob) calls it. The elevator, then, moves two times (either goes up or goes down). What does it should do in those movements?

- Elevator starts on 4th floor
- 2 people call at the same time:
    - Call: Alice (40kg) from 3rd to 0 floor
    - Call: Angelina (40kg) from 2nd to 1st floor
- Elevator moves until it picks Angelina up.
- New call when the elevator is in the 2nd floor: Bob from 4th to 0 floor
- Elevator moves 2 times

Task Description

4) Try to understand how the features interact. In this case, **you should leverage the Varviz plug-in** when executing the tesrt.
5) Describe what you understand from the interactions presented in the graph.
6) Are there any suspicious interaction? Why?

Group 4

Schedule

13. Warm-up 1: VarXplorer - WordPress
14. Task 1: VarXplorer - Elevator
15. Warm-up 2: Varviz - Wordpress
16. Task 2: Varviz - Telephone

Warm-up: VarXplorer - WordPress

Steps:

- **Feature interaction definition**
- **Experiment objective: identifying bugs related to feature interactions**
- **Introduction to VarXplorer: features and relationships (suppress and enable)**
- **Example based on the WordPress**

WordPress main Features:

- **Smiley**: replaces characters with smiley faces.
- **Fahrenheit**: displays temperature in Fahrenheit
- **Weather**: presents the current weather in either Celsius or Fahrenheit

Test case scenario:

- you are given a WordPress test case that shows the current temperature to the user.

Task Description

1) Try to understand how the features interact. In this case, **you should leverage the VarXplorer plug-in** when executing the test case.
2) Describe what you understand from the interactions presented in the graph.
3) Is there any suspicious interaction? Something that does not agree with the requirements? Why?

Task #1 VarXplorer - ELEVATOR

You have at most 30 minutes to try understanding the interactions and tell us whether they represent a bug or not.

Elevator: It is an elevator model that is extensible by various features such as stopping if the elevator is empty or priority service for a special floor.

Elevator main Features:

- **2/3- full**: When the lift detects that it is more than two-thirds full, it does not stop in response to landing calls, since it is unlikely to be able to accept more passengers. Instead, it gives priority to passengers already inside the lift, as serving them will help reduce its load.
- **Overloaded**: When the lift is overloaded, the doors will not close. Some passengers must get out.
- **Executive floor**: The lift gives priority to calls from the executive floor.
- **Weight**: Updates the weight when someone gets inside or gets out.

Program details:

- ***Timeshift*** method: it executes one movement per time: either moves up or moves down or opens the door (someone gets in) or closes the door (and moves)
- ***respectFloorCalls*** variable (related to the feature **2/3- full)**:
    - Boolean value that when false the elevator does not attend the calls until the weight is less than 2/3.
- Additional information:
    - **Max Weight: 100kg**
    - **Executive floor: 4th**
    - **2/3 full: above 67kg**

Test case scenario:

Three people need to use the elevator. The first two (Alice and Angeline) call at the same time when the elevator is idle at the 4th floor. Then, right after it picks the second girl up and closes the door at the 2nd floor, the third person (Bob) calls the elevator. The elevator, then, moves two times (either goes up or goes down). What does it should do in those movements?

- Elevator starts on 4th floor
- 2 people call at the same time:
    - Call: Alice (40kg) from 3rd to 0 floor
    - Call: Angelina (40kg) from 2nd to 1st floor
- Elevator moves until it picks Angelina up.
- New call when the elevator is in the 2nd floor: Bob from 4th to 0 floor
- Elevator moves 2 times. What does it should do?

Task Description

1) Try to understand how the features interact. In this case, **you should leverage the VarXplorer plug-in** when executing the tesrt.
2) Describe what you understand from the interactions presented in the graph.
3) Are there any suspicious interaction? Why?

Warm-up: Varviz - WordPress

Goals:
- **Introduction to Varviz: trace (diamonds, expressions, statements, variables)**
- **Example based on the WordPress**

WordPress main Features:
- **Smiley**: replaces characters to smile faces.
- **Fahrenheit**: temperature is displayed in Fahrenheit
- **Weather**: presents the current weather in either Celsius or Fahrenheit

Test case scenario:

- The program shows the current temperature to the user.

Task Description

1) Try to understand how the features interact. In this case, **you should leverage the Varviz plug-in** when execute the tasks.
2) Describe what you understand from the interactions presented in the graph.
3) Is there any suspicious interaction? Why?

Task #2 Varviz - TELEPHONE SYSTEM

You have at most 30 minutes to try understanding the interactions and tell us whether they represent a bug or not.

Telephone main Features:

- **Call forwarding busy line (CFB):** All calls to the subscribing line are redirected to a predetermined number when the line is busy.
- **Call Waiting**: allows the subscriber to be noticed that another party (incoming call) is trying to reach his number while her line is busy, and to accept the new call by placing the original call on hold.

Program details:

- ***Timeshift()*** method**:** executes one action per time (eg., setting caller as off/on hook, starting ringing)
- ***disconnectCaller()*** method**:** disconnects the caller (putting as on hook)
- ***Call.onhold*** variable**:** true when call is on waiting (controlled by the feature **Call Waiting**)

Test case scenario:

The system receives a call from Alice to Sophie. Unfortunately, Sophie is busy and cannot receive the call. Sophie has enabled the *call waiting* option to be executed when she is busy.

- Alice calls to Sophie
- Sophie is busy
- Sophie is subscribed to call waiting.
- Sophie did not choose any person to the call be forwarded.

Task Description

7) Try to understand how the features interact. In this case, **you should leverage the Varviz plug-in** when execute the tasks.
8) Describe what you understand from the interactions you can perceive in the trace.
9) Is there any suspicious interaction? Why?

B.3 EVALUATION: THE TIME MEASURED FOR THE PARTICIPANTS

Group	OrderTool	OrderSys	System	Tool	Time
GEZ	VarvizFirst	ElevFirst	Elevator	Varviz	666
GEZ	VarvizFirst	ElevFirst	Elevator	Varviz	661
GEZ	VarvizFirst	ElevFirst	Elevator	Varviz	600
GEZ	VarvizFirst	ElevFirst	Elevator	Varviz	705
GEZ	VarvizFirst	ElevFirst	Elevator	Varviz	463
GEZ	VarvizSecond	ElevSecond	Elevator	Varviz	467
GEZ	VarvizSecond	ElevSecond	Elevator	Varviz	435
GEZ	VarvizSecond	ElevSecond	Elevator	Varviz	457
GEZ	VarvizSecond	ElevSecond	Elevator	Varviz	596
GEZ	VarvizSecond	ElevSecond	Elevator	Varviz	430
GEZ	VarvizFirst	ElevFirst	Elevator	Varviz	763
GEZ	VarvizSecond	ElevSecond	Elevator	Varviz	489
GEX	VarXpSecond	ElevSecond	Elevator	VarXp	220
GEX	VarXpSecond	ElevSecond	Elevator	VarXp	216
GEX	VarXpSecond	ElevSecond	Elevator	VarXp	172
GEX	VarXpSecond	ElevSecond	Elevator	VarXp	130
GEX	VarXpSecond	ElevSecond	Elevator	VarXp	261
GEX	VarXpFirst	ElevFirst	Elevator	VarXp	196
GEX	VarXpFirst	ElevFirst	Elevator	VarXp	264
GEX	VarXpFirst	ElevFirst	Elevator	VarXp	154
GEX	VarXpFirst	ElevFirst	Elevator	VarXp	287
GEX	VarXpFirst	ElevFirst	Elevator	VarXp	259
GEX	VarXpSecond	ElevSecond	Elevator	VarXp	208
GEX	VarXpFirst	ElevFirst	Elevator	VarXp	133
GTZ	VarvizFirst	TelepFirst	Telephone	Varviz	778
GTZ	VarvizFirst	TelepFirst	Telephone	Varviz	500
GTZ	VarvizFirst	TelepFirst	Telephone	Varviz	649
GTZ	VarvizFirst	TelepFirst	Telephone	Varviz	601
GTZ	VarvizFirst	TelepFirst	Telephone	Varviz	615
GTZ	VarvizSecond	TelepSecond	Telephone	Varviz	580
GTZ	VarvizSecond	TelepSecond	Telephone	Varviz	724
GTZ	VarvizSecond	TelepSecond	Telephone	Varviz	549
GTZ	VarvizSecond	TelepSecond	Telephone	Varviz	457
GTZ	VarvizSecond	TelepSecond	Telephone	Varviz	709
GTZ	VarvizFirst	TelepFirst	Telephone	Varviz	519
GTZ	VarvizSecond	TelepSecond	Telephone	Varviz	526
GTX	VarXpSecond	TelepSecond	Telephone	VarXp	281
GTX	VarXpSecond	TelepSecond	Telephone	VarXp	183
GTX	VarXpSecond	TelepSecond	Telephone	VarXp	185

GTX	VarXpSecond	TelepSecond	Telephone	VarXp	157
GTX	VarXpSecond	TelepSecond	Telephone	VarXp	120
GTX	VarXpFirst	TelepFirst	Telephone	VarXp	139
GTX	VarXpFirst	TelepFirst	Telephone	VarXp	197
GTX	VarXpFirst	TelepFirst	Telephone	VarXp	123
GTX	VarXpFirst	TelepFirst	Telephone	VarXp	119
GTX	VarXpFirst	TelepFirst	Telephone	VarXp	180
GTX	VarXpSecond	TelepSecond	Telephone	VarXp	134
GTX	VarXpFirst	TelepFirst	Telephone	VarXp	241

Table B.1: The time measured for the participants in the evaluation

B.4 EVALUATION: R SCRIPT

Listing B.1: R script used in the evaluation of VarXplorer

```
library(dplyr)
library(sjstats)

#varXplorer experiment
#author: Larissa Rocha

###variables
eleViz <- dataCompleteG%>%filter(System=="Elevator"&Tool=="Varviz")
eleVax <- dataCompleteG%>%filter(System=="Elevator"&Tool=="VarXp")
telViz <- dataCompleteG%>%filter(System=="Telephone"&Tool=="Varviz")
telVax <- dataCompleteG%>%filter(System=="Telephone"&Tool=="VarXp")
timeVarviz <-dataCompleteG%>%filter(Tool=="Varviz")
timeVarX <-dataCompleteG%>%filter(Tool=="VarXp")
varvizFirst <- dataCompleteG%>%filter(OrderTool=="VarvizFirst")
varvizSecond <- dataCompleteG%>%filter(OrderTool=="VarvizSecond")
varXFirst <- dataCompleteG%>%filter(OrderTool=="VarXpFirst")
varXSecond <- dataCompleteG%>%filter(OrderTool=="VarXpSecond")

#standard deviation
sd(dataCompleteG$Time[1:12])
sd(dataCompleteG$Time[13:24])
sd(dataCompleteG$Time[25:36])
sd(dataCompleteG$Time[37:48])

#mean
mean(eleVax$Time)
mean(telVax$Time)
mean(eleViz$Time)
mean(telViz$Time)

mean(timeVarviz$Time)
mean(timeVarX$Time)


```

```
###-----comparing Varvix x VarXplorer in general
boxplot(dataGroupoOrdem$Time ~ dataGroupoOrdem$Tool , col=c("gray"), ylab="Time␣
    (seconds)", xlab="Tool␣used")

###-----test to see if the order has influence
kruskal.test(dataGroupoOrdem$Time~dataGroupoOrdem$Group)
boxplot(dataGroupoOrdem$Time~dataGroupoOrdem$Group , ylab="Time␣(seconds)",
    xlab="Groups")
kruskal.test(dataCompleteG$Time~dataCompleteG$OrderTool)
kruskal.test(dataCompleteG$Time~dataCompleteG$OrderSys)



###-----Shapiro-Wilk normality test
shapiro.test(timeVarviz$Time)
shapiro.test(timeVarX$Time)

#normal per subgroup
shapiro.test(dataCompleteG$Time[1:12])
shapiro.test(dataCompleteG$Time[13:24])
shapiro.test(dataCompleteG$Time[25:36])
shapiro.test(dataCompleteG$Time[37:48])

###----Bartlett test of homogeneity of variances
bartlett.test(dataCompleteG$Time~dataCompleteG$Tool)
bartlett.test(dataCompleteG$Time~dataCompleteG$System)

bartlett.test(dataCompleteG$Time~dataCompleteG$Group)
boxplot(dataCompleteG$Time~dataCompleteG$Group , col=c("gray"), ylab="Time␣(seconds)",
    xlab="System␣versus␣Tool")
bartlett.test(dataCompleteG$Time~dataCompleteG$Tool)
boxplot(dataCompleteG$Time~dataCompleteG$Tool)


###Variance homogeneity by order
bartlett.test(dataCompleteG$Time~dataCompleteG$OrderTool)
boxplot(dataCompleteG$Time~dataCompleteG$OrderTool)
bartlett.test(dataCompleteG$Time~dataCompleteG$OrderSys)
boxplot(dataCompleteG$Time~dataCompleteG$OrderSys)


#normal by order
shapiro.test(varvizFirst$Time)
shapiro.test(varvizSecond$Time)
shapiro.test(varXFirst$Time)
shapiro.test(varXSecond$Time)

##--- ANOVA test
#anov = aov(Time~System+Tool+Group)
anov = aov(Time~System+Tool)
summary(anov)
shapiro.test(resid(anov))
bartlett.test(dataCompleteG$Time~dataCompleteG$Tool)
cohens_f(anov)
eta_sq(anov)
anova_stats(anov)
#TukeyHSD(anov)


##---testing the learning effect (order)
boxplot(dataCompleteG$Time~dataCompleteG$OrderSys ,  ylab="Time␣in␣seconds",
    xlab="Systems␣Order")
boxplot(dataCompleteG$Time~dataCompleteG$OrderTool ,  ylab="Time␣in␣seconds",
    xlab="Tools␣Order")

anov1 = aov(Time~OrderTool)
```

```
summary(anov1)
TukeyHSD(anov1)
shapiro.test(resid(anov1))
bartlett.test(dataCompleteG$Time~dataCompleteG$OrderTool)

anov2 = aov(Time~OrderSys)
summary(anov2)
TukeyHSD(anov2)
shapiro.test(resid(anov2))
bartlett.test(dataCompleteG$Time~dataCompleteG$OrderSys)


#----other analysis-----

#no-parametric inference/analysis
kruskal.test(Time~Group)
wilcox.test(Time~Tool)
wilcox.test(Time~System)

###comparing per system
boxplot(eleViz$Time,telViz$Time)
boxplot(eleVax$Time,telVax$Time)

###comparing per tool
boxplot(eleViz$Time,eleVax$Time)
boxplot(telViz$Time,telVax$Time)
```

Appendix C

EXPLORATORY STUDY: DATA AND SUPPORT MATERIAL

C.1 ALL SPECIFICATIONS CREATED DURING THE EXPLORATORY STUDY

```

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
    <system name="RiSE_Event_SPL">
        <specification type="Allow"><require from="Activity" to="RegistrationSpeakerActivity"> <var
            name="int_ActivitySpeaker.idActivity"/> </require></specification>
        <specification type="Allow"><require from="RegistrationSpeakerActivity" to="Activity"><var
            name="int_ActivitySpeaker.idActivity"/> </require></specification>
        <specification type="Allow"><require from="RegistrationSpeakerActivity" to="Activity"><var
            name="int_ActivitySpeaker.idUser"/></require></specification>
        <specification type="Allow"><require from="RegistrationSpeakerActivity" to="Speaker"><var
            name="int_ActivitySpeaker.idActivity"/></require></specification>
        <specification type="Allow"><require from="RegistrationSpeakerActivity" to="Speaker"><var
            name="int_ActivitySpeaker.idUser"/></require></specification>
        <specification type="Allow"><require from="Speaker" to="RegistrationSpeakerActivity"><var
            name="int_ActivitySpeaker.idUser"/></require></specification>
        <specification type="Allow"><require from="RegistrationSpeakerActivity" to="Speaker"><var
            name="Speaker_e"/></require></specification>
        <specification type="Allow"><require from="RegistrationSpeakerActivity" to="Speaker"><var
            name="Speaker_newSpeaker"/></require></specification>
        <specification type="Allow"><require from="RegistrationSpeakerActivity" to="Speaker"><var
            name="Speaker_newSpeaker"/></require></specification>
        <specification type="Allow"><require from="RegistrationSpeakerActivity" to="Speaker"><var
            name="Speaker_newSpeaker"/></require></specification>
        <specification type="Allow"><require from="RegistrationSpeakerActivity" to="Speaker"><var
            name="Exception_e"/></require></specification>
        <specification type="Allow"><require from="RegistrationSpeakerActivity" to="Speaker"><var
            name="Exception_e"/></require></specification>
        <specification type="Allow"><require from="Speaker" to="RegistrationSpeakerActivity"><var
            name="Speaker_newSpeaker"/></require></specification>
        <specification type="Allow"><require from="User" to="Speaker"><var name="int_
            usersSize"/></require></specification><
         specification type="Allow"><require from="User" to="Speaker"><var name="String_
            Speaker.biography"/></require></specification>
        <specification type="Allow"><require from="User" to="Speaker"><var name="int_
            User.idUser"/></require></specification>
        <specification type="Allow"><require from="User" to="Speaker"><var name="String_
            User.email"/></require></specification>
        <specification type="Allow"><require from="User" to="Speaker"><var name="String_
            User.filiation"/></require></specification>
        <specification type="Allow"><require from="User" to="Speaker"><var name="String_
            User.nameUser"/></require></specification>
        <specification type="Allow"><require from="User" to="Speaker"><var name="String_
            User.password"/></require></specification>
        <specification type="Allow"><require from="User" to="Speaker"><var name="User$TypeUser_
            User.typeUser"/></require></specification>
        <specification type="Allow"><require from="User" to="Speaker"><var name="Speaker_
            e"/></require></specification>
        <specification type="Allow"><require from="User" to="Speaker"><var name="Speaker_
            newSpeaker"/></require></specification>
        <specification type="Allow"><require from="User" to="Speaker"><var name="Speaker_
            newSpeaker"/></require></specification>
        <specification type="Allow"><require from="User" to="Speaker"><var name="Exception␣
            e"/></require></specification>
        <specification type="Allow"><require from="Speaker" to="User"><var name="int␣
            usersSize"/></require></specification>
        <specification type="Allow"><require from="Speaker" to="User"><var name="String␣
            Speaker.biography"/></require></specification>
        <specification type="Allow"><require from="Speaker" to="User"><var name="int␣
            User.idUser"/></require></specification>
        <specification type="Allow"><require from="Speaker" to="User"><var name="String␣
            User.email"/></require></specification>
        <specification type="Allow"><require from="Speaker" to="User"><var name="String␣
            User.filiation"/></require></specification>
        <specification type="Allow"><require from="Speaker" to="User"><var name="String␣
            User.nameUser"/></require></specification>
        <specification type="Allow"><require from="Speaker" to="User"><var name="String␣
            User.password"/></require></specification>
        <specification type="Allow"><require from="Speaker" to="User"><var name="User$TypeUser␣
            User.typeUser"/></require></specification>
        <specification type="Allow"><require from="Speaker" to="User"><var name="Speaker␣
            e"/></require></specification>
        <specification type="Allow"><require from="Speaker" to="User"><var name="Speaker␣
            e"/></require></specification>
        <specification type="Allow"><require from="Speaker" to="User"><var name="Speaker␣
            newSpeaker"/></require></specification>
        <specification type="Allow"><require from="Speaker" to="User"><var name="Speaker␣
            newSpeaker"/></require></specification>
        <specification type="Allow"><require from="Speaker" to="User"><var name="Exception␣
            e"/></require></specification>
        <specification type="Allow"><require from="RegistrationSpeakerActivity" to="User"><var
            name="Speaker␣e"/></require></specification>
        <specification type="Allow"><require from="RegistrationSpeakerActivity" to="User"><var
            name="Speaker␣newSpeaker"/></require></specification>
        <specification type="Allow"><require from="RegistrationSpeakerActivity" to="User"><var
            name="Exception␣e"/></require></specification>
        <specification type="Allow"><require from="User" to="RegistrationSpeakerActivity"><var
            name="Speaker␣newSpeaker"/></require></specification>
        <specification type="Allow"><require from="User" to="Speaker"><var name="int␣
            ActivitySpeaker.idUser"/></require></specification>
        <specification type="Allow"><require from="User" to="RegistrationSpeakerActivity"><var
            name="int␣ActivitySpeaker.idUser"/></require></specification>
        <specification type="Allow"><suppress from="Speaker" to="RegistrationSpeakerActivity"><var
            name="boolean␣answer"/></suppress></specification>
        <specification type="Allow"><require from="RegistrationSpeakerActivity" to="Activity"><var
            name="boolean␣answer"/></require></specification>
        <specification type="Allow"><suppress from="RegistrationSpeakerActivity" to="Activity"><var
            name="boolean␣answer"/></suppress></specification>
        <specification type="Allow"><require from="Speaker" to="Activity"><var name="boolean␣
            answer"/></require></specification>
        <specification type="Allow"><require from="Speaker" to="Activity"><var name="int␣
            ActivitySpeaker.idUser"/></require></specification>
        <specification type="Allow"><require from="Speaker" to="Activity"><var name="int␣
            ActivitySpeaker.idActivity"/> </require> </specification>
        <specification type="Allow"><suppress from="Speaker" to="Activity"><var name="boolean␣
            answer"/></suppress></specification>
        <specification type="Allow"><require from="Speaker" to="RegistrationSpeakerActivity"><var
            name="boolean␣answer"/></require></specification>
        <specification type="Allow"><require from="Activity" to="Speaker"><var name="int␣
            ActivitySpeaker.idActivity"/></require></specification>
        <specification type="Allow"><require from="User" to="Reviewer"><var name="int␣
            usersSize"/></require></specification>
        <specification type="Allow"><require from="User" to="Reviewer"><var name="int␣
            User.idUser"/></require></specification>
        <specification type="Allow"><require from="Reviewer" to="User"><var name="int␣
            usersSize"/></require></specification>
        <specification type="Allow"><require from="CompleteSubmission" to="Activity"><var name="int␣
            Submission.idActivity"/></require></specification>
        <specification type="Allow"><require from="CompleteSubmission" to="Activity"><var name="int␣
            SubmissionUser.idActivity"/></require></specification>
        <specification type="Allow"><require from="PartialSubmission" to="Activity"><var name="int␣
            Submission.idActivity"/></require></specification>
        <specification type="Allow"><require from="PartialSubmission" to="Activity"><var name="int␣
            SubmissionUser.idActivity"/></require></specification>
        <specification type="Allow"><require from="CompleteSubmission" to="User"><var name="int␣
            idUsuario"/></require></specification>
        <specification type="Allow"><require from="CompleteSubmission" to="User"><var name="int␣
            SubmissionUser.idUser"/></require></specification>
        <specification type="Allow"><require from="PartialSubmission" to="User"><var name="int␣
            idUsuario"/></require></specification>
        <specification type="Allow"><require from="PartialSubmission" to="User"><var name="int␣
            SubmissionUser.idUser"/></require></specification>
        <specification type="Allow"><suppress from="PartialSubmission" to="CompleteSubmission"><var
            name="SubmissionRepositoryNewBase␣
            SubmissionRepositoryNewBase.instance"/></suppress></specification>
        <specification type="Allow"><suppress from="PartialSubmission" to="CompleteSubmission"><var
            name="SubmissionRepository␣submissionRepository"/></suppress></specification>
        <specification type="Allow"><suppress from="PartialSubmission" to="CompleteSubmission"><var
            name="SubmissionControl␣RiSEventFacade.submissions"/></suppress></specification>
        <specification type="Allow"><suppress from="PartialSubmission" to="CompleteSubmission"><var
            name="SubmissionUserRepositoryNewBase␣
            SubmissionUserRepositoryNewBase.instance"/></suppress></specification>
        <specification type="Allow"><suppress from="PartialSubmission" to="CompleteSubmission"><var name="SubmissionUserControl_RiSEventFacade.submissionUsers"/></suppress></specification>
        <specification type="Allow"><suppress from="PartialSubmission" to="CompleteSubmission"><var name="int_lastsub"/></suppress></specification>
        <specification type="Allow"><suppress from="PartialSubmission" to="CompleteSubmission"><var name="int_Test3.idSubmission"/></suppress></specification>
        <specification type="Allow"><suppress from="PartialSubmission" to="CompleteSubmission"><var name="Submission_submission"/></suppress></specification>
        <specification type="Allow"><suppress from="PartialSubmission" to="CompleteSubmission"><var name="int_Submission.idActivity"/></suppress></specification>
        <specification type="Allow"><suppress from="PartialSubmission" to="CompleteSubmission"><var name="int_idUsuario"/></suppress></specification>
        <specification type="Allow"><suppress from="PartialSubmission" to="CompleteSubmission"><var name="int_SubmissionUser.idUser"/></suppress></specification>
        <specification type="Allow"><suppress from="PartialSubmission" to="CompleteSubmission"><var name="int_SubmissionUser.idActivity"/></suppress></specification>
        <specification type="Allow"><suppress from="CompleteSubmission" to="PartialSubmission"><var name="SubmissionRepositoryNewBase_SubmissionRepositoryNewBase.instance"/></suppress></specification>
        <specification type="Allow"><suppress from="CompleteSubmission" to="PartialSubmission"><var name="SubmissionRepository_submissionRepository"/></suppress></specification>
        <specification type="Allow"><suppress from="CompleteSubmission" to="PartialSubmission"><var name="SubmissionControl_RiSEventFacade.submissions"/></suppress></specification>
        <specification type="Allow"><suppress from="CompleteSubmission" to="PartialSubmission"><var name="SubmissionUserRepositoryNewBase_SubmissionUserRepositoryNewBase.instance"/></suppress></specification>
        <specification type="Allow"><suppress from="CompleteSubmission" to="PartialSubmission"><var name="SubmissionUserControl_RiSEventFacade.submissionUsers"/></suppress></specification>
        <specification type="Allow"><suppress from="CompleteSubmission" to="PartialSubmission"><var name="int_lastsub"/></suppress></specification>
        <specification type="Allow"><suppress from="CompleteSubmission" to="PartialSubmission"><var name="int_Test3.idSubmission"/></suppress></specification>
        <specification type="Allow"><suppress from="CompleteSubmission" to="PartialSubmission"><var name="Submission_submission"/></suppress></specification>
        <specification type="Allow"><suppress from="CompleteSubmission" to="PartialSubmission"><var name="int_Submission.idActivity"/></suppress></specification>
        <specification type="Allow"><suppress from="CompleteSubmission" to="PartialSubmission"><var name="int_idUsuario"/></suppress></specification>
        <specification type="Allow"><suppress from="CompleteSubmission" to="PartialSubmission"><var name="int_SubmissionUser.idUser"/></suppress></specification>
        <specification type="Allow"><suppress from="CompleteSubmission" to="PartialSubmission"><var name="int_SubmissionUser.idActivity"/></suppress></specification>
        <specification type="Allow"><require from="PartialSubmission" to="User"><var name="int_idCorrespondingAuthor"/></require></specification>
        <specification type="Allow"><require from="PartialSubmission" to="User"><var name="int_SubmissionAuthor.idAuthor"/></require></specification>
        <specification type="Allow"><require from="CompleteSubmission" to="User"><var name="int_idCorrespondingAuthor"/></require></specification>
        <specification type="Allow"><require from="CompleteSubmission" to="User"><var name="int_SubmissionAuthor.idAuthor"/></require></specification>
        <specification type="Allow"><suppress from="CompleteSubmission" to="PartialSubmission"><var name="SubmissionAuthorRepositoryNewBase_SubmissionAuthorRepositoryNewBase.instance"/></suppress></specification>
        <specification type="Allow"><suppress from="CompleteSubmission" to="PartialSubmission"><var name="SubmissionAuthorControl_RiSEventFacade.submissionAuthors"/></suppress></specification>
        <specification type="Allow"><suppress from="CompleteSubmission" to="PartialSubmission"><var name="int_idCorrespondingAuthor"/></suppress></specification>
        <specification type="Allow"><suppress from="CompleteSubmission" to="PartialSubmission"><var name="int_SubmissionAuthor.idAuthor"/></suppress></specification>
        <specification type="Allow"><suppress from="CompleteSubmission" to="PartialSubmission"><var name="int_SubmissionAuthor.idActivity"/></suppress></specification>
        <specification type="Allow"><suppress from="PartialSubmission" to="CompleteSubmission"><var name="SubmissionAuthorRepositoryNewBase_SubmissionAuthorRepositoryNewBase.instance"/></suppress></specification>
        <specification type="Allow"><suppress from="PartialSubmission" to="CompleteSubmission"><var name="SubmissionAuthorControl_RiSEventFacade.submissionAuthors"/></suppress></specification>
        <specification type="Allow"><suppress from="PartialSubmission" to="CompleteSubmission"><var name="int_idCorrespondingAuthor"/></suppress></specification>
        <specification type="Allow"><suppress from="PartialSubmission" to="CompleteSubmission"><var name="int_SubmissionAuthor.idAuthor"/></suppress></specification>
        <specification type="Allow"><suppress from="PartialSubmission" to="CompleteSubmission"><var name="int_SubmissionAuthor.idActivity"/></suppress></specification>
        <specification type="Allow"><require from="CompleteSubmission" to="Activity"><var name="int_SubmissionAuthor.idActivity"/></require></specification>
        <specification type="Allow"><require from="PartialSubmission" to="Activity"><var name="int_SubmissionAuthor.idActivity"/></require></specification>
        <specification type="Allow"><require from="CompleteSubmission" to="InsertAuthor"><var name="SubmissionAuthorRepositoryNewBase_SubmissionAuthorRepositoryNewBase.instance"/></require></specification>
        <specification type="Allow"><require from="CompleteSubmission" to="InsertAuthor"><var name="SubmissionAuthorControl_RiSEventFacade.submissionAuthors"/></require></specification>
        <specification type="Allow"><require from="CompleteSubmission" to="InsertAuthor"><var name="int_idCorrespondingAuthor"/></require></specification>
        <specification type="Allow"><require from="CompleteSubmission" to="InsertAuthor"><var name="int_SubmissionAuthor.idAuthor"/></require></specification>
        <specification type="Allow"><require from="CompleteSubmission" to="InsertAuthor"><var name="int_SubmissionAuthor.idActivity"/></require></specification>
```

```
<specification type="Allow"><require from="InsertAuthor" to="Activity"><var name="int␣
     SubmissionAuthor.idActivity"/></require></specification>
<specification type="Allow"><require from="User" to="InsertAuthor"><var name="int␣
     idCorrespondingAuthor"/></require></specification>
<specification type="Allow"><require from="User" to="InsertAuthor"><var name="int␣
     SubmissionAuthor.idAuthor"/></require></specification>
<specification type="Allow"><require from="InsertAuthor" to="User"><var name="int␣
     idCorrespondingAuthor"/></require></specification>
<specification type="Allow"><require from="InsertAuthor" to="User"><var name="int␣
     SubmissionAuthor.idAuthor"/></require></specification>
<specification type="Allow"><require from="Activity" to="InsertAuthor"><var name="int␣
     SubmissionAuthor.idActivity"/></require></specification>
<specification type="Allow"><require from="PartialSubmission" to="InsertAuthor"><var
     name="SubmissionAuthorRepositoryNewBase␣
     SubmissionAuthorRepositoryNewBase.instance"/></require></specification>
<specification type="Allow"><require from="PartialSubmission" to="InsertAuthor"><var
     name="SubmissionAuthorControl␣
     RiSEventFacade.submissionAuthors"/></require></specification><specification
     type="Allow"><require from="PartialSubmission" to="InsertAuthor"><var name="int␣
     idCorrespondingAuthor"/></require></specification>
<specification type="Allow"><require from="PartialSubmission" to="InsertAuthor"><var
     name="int␣SubmissionAuthor.idAuthor"/></require></specification>
<specification type="Allow"><require from="PartialSubmission" to="InsertAuthor"><var
     name="int␣SubmissionAuthor.idActivity"/></require></specification>
<specification type="Allow"><require from="Reviewer" to="User"><var name="String␣
     User.email"/></require></specification>
<specification type="Allow"><require from="Reviewer" to="User"><var name="String␣
     User.filiation"/></require></specification>
<specification type="Allow"><require from="Reviewer" to="User"><var name="String␣
     User.nameUser"/></require></specification>
<specification type="Allow"><require from="Reviewer" to="User"><var name="String␣
     User.password"/></require></specification>
<specification type="Allow"><require from="Reviewer" to="User"><var name="User$TypeUser␣
     User.typeUser"/></require></specification>
<specification type="Allow"><require from="Reviewer" to="User"><var name="String␣
     Reviewer.knowledgeArea"/></require></specification>
<specification type="Allow"><require from="User" to="Reviewer"><var name="String␣
     User.email"/></require></specification>
<specification type="Allow"><require from="User" to="Reviewer"><var name="String␣
     User.filiation"/></require></specification>
<specification type="Allow"><require from="User" to="Reviewer"><var name="String␣
     User.nameUser"/></require></specification>
<specification type="Allow"><require from="User" to="Reviewer"><var name="String␣
     User.password"/></require></specification>

<specification type="Allow"><require from="User" to="Reviewer"><var name="User$TypeUser␣
     User.typeUser"/></require></specification><specification type="Allow"><require
     from="User" to="Reviewer"><var name="String␣
     Reviewer.knowledgeArea"/></require></specification>
<specification type="Allow"><suppress from="PartialSubmission" to="CompleteSubmission"><var
     name="int␣Test4.idSubmission"/></suppress></specification>
<specification type="Allow"><suppress from="PartialSubmission" to="CompleteSubmission"><var
     name="int␣Review.idSubmission"/></suppress></specification>
<specification type="Allow"><suppress from="PartialSubmission" to="CompleteSubmission"><var
     name="int␣Review.round"/></suppress></specification>
<specification type="Allow"><suppress from="CompleteSubmission" to="PartialSubmission"><var
     name="int␣Test4.idSubmission"/></suppress></specification>
<specification type="Allow"><suppress from="CompleteSubmission" to="PartialSubmission"><var
     name="int␣Review.idSubmission"/></suppress></specification>
<specification type="Allow"><suppress from="CompleteSubmission" to="PartialSubmission"><var
     name="int␣Review.round"/></suppress></specification>
<specification type="Allow"><require from="PartialSubmission" to="Review"><var name="int␣
     Review.idSubmission"/></require></specification>
<specification type="Allow"><require from="PartialSubmission" to="Review"><var name="int␣
     Review.round"/></require></specification>
<specification type="Allow"><require from="CompleteSubmission" to="Review"><var name="int␣
     Review.idSubmission"/></require></specification>
<specification type="Allow"><require from="CompleteSubmission" to="Review"><var name="int␣
     Review.round"/></require></specification>
<specification type="Allow"><suppress from="CompleteSubmission" to="PartialSubmission"><var
     name="Review␣r"/></suppress></specification>
<specification type="Allow"><suppress from="CompleteSubmission" to="PartialSubmission"><var
     name="int␣round"/></suppress></specification>
<specification type="Allow"><suppress from="PartialSubmission" to="CompleteSubmission"><var
     name="Review␣r"/></suppress></specification>
<specification type="Allow"><suppress from="PartialSubmission" to="CompleteSubmission"><var
     name="int␣round"/></suppress></specification>
<specification type="Allow"><require from="PartialSubmission" to="Review"><var name="Review␣
     r"/></require></specification>
<specification type="Allow"><require from="PartialSubmission" to="Review"><var name="int␣
     round"/></require></specification>
<specification type="Allow"><require from="CompleteSubmission" to="Review"><var name="Review␣
     r"/></require></specification>
<specification type="Allow"><require from="CompleteSubmission" to="Review"><var name="int␣
     round"/></require></specification>
<specification type="Allow"><require from="CompleteSubmission" to="Reviewer"><var name="int␣
     round"/></require></specification>
<specification type="Allow"><require from="Reviewer" to="Review"><var name="int␣
     usersSize"/></require></specification>
```

```
    <specification type="Allow"><require from="Reviewer" to="Review"><var name="int␣
        User.idUser"/></require></specification>
    <specification type="Allow"><require from="Reviewer" to="Review"><var name="String␣
        User.email"/></require></specification>
    <specification type="Allow"><require from="Reviewer" to="Review"><var name="String␣
        User.filiation"/></require></specification>
    <specification type="Allow"><require from="Reviewer" to="Review"><var name="String␣
        User.nameUser"/></require></specification>
    <specification type="Allow"><require from="Reviewer" to="Review"><var name="String␣
        User.password"/></require></specification>
    <specification type="Allow"><require from="Reviewer" to="Review"><var name="User$TypeUser␣
        User.typeUser"/></require></specification>
    <specification type="Allow"><require from="Reviewer" to="Review"><var name="String␣
        Reviewer.knowledgeArea"/></require></specification>
    <specification type="Allow"><require from="Reviewer" to="Review"><var name="int␣
        Review.round"/></require></specification>
    <specification type="Allow"><require from="Reviewer" to="Review"><var name="Review␣
        r"/></require></specification>
    <specification type="Allow"><require from="Reviewer" to="Review"><var name="int␣
        round"/></require></specification>
    <specification type="Allow"><require from="Review" to="Reviewer"><var name="int␣
        usersSize"/></require></specification>
    <specification type="Allow"><require from="Review" to="Reviewer"><var name="int␣
        User.idUser"/></require></specification>
    <specification type="Allow"><require from="Review" to="Reviewer"><var name="String␣
        User.email"/></require></specification>
    <specification type="Allow"><require from="Review" to="Reviewer"><var name="String␣
        User.filiation"/></require></specification>
    <specification type="Allow"><require from="Review" to="Reviewer"><var name="String␣
        User.nameUser"/></require></specification>
    <specification type="Allow"><require from="Review" to="Reviewer"><var name="String␣
        User.password"/></require></specification>
    <specification type="Allow"><require from="Review" to="Reviewer"><var name="User$TypeUser␣
        User.typeUser"/></require></specification>
    <specification type="Allow"><require from="Review" to="Reviewer"><var name="String␣
        Reviewer.knowledgeArea"/></require></specification>
    <specification type="Allow"><require from="Review" to="Reviewer"><var name="int␣
        round"/></require></specification>
    <specification type="Allow"><require from="PartialSubmission" to="Reviewer"><var name="int␣
        round"/></require></specification>
    <specification type="Allow"><require from="Activity" to="Registration"><var name="int␣
        Registration.idEvent"/></require></specification>
    <specification type="Allow"><require from="Activity" to="Registration"><var name="int␣
        ActivityUser.idUser"/></require></specification>
    <specification type="Allow"><require from="Registration" to="Activity"><var name="int␣
        Registration.idEvent"/></require></specification>
    <specification type="Allow"><require from="Registration" to="Activity"><var name="int␣
        ActivityUser.idUser"/></require></specification>
    <specification type="Allow"><require from="RegistrationUserActivity" to="Activity"><var
        name="int␣Registration.idEvent"/></require></specification>
    <specification type="Allow"><require from="RegistrationUserActivity" to="Activity"><var
        name="int␣ActivityUser.idActivity"/></require></specification>
    <specification type="Allow"><require from="RegistrationUserActivity" to="Activity"><var
        name="int␣ActivityUser.idUser"/></require></specification>
    <specification type="Allow"><require from="Activity" to="User"><var name="int␣
        ActivityUser.idUser"/></require></specification>
    <specification type="Allow"><require from="User" to="Activity"><var name="int␣
        ActivityUser.idUser"/></require></specification>
    <specification type="Allow"><require from="RegistrationUserActivity" to="User"><var name="int␣
        Registration.idUser"/></require></specification>
    <specification type="Allow"><require from="RegistrationUserActivity" to="User"><var name="int␣
        ActivityUser.idUser"/></require></specification>
    <specification type="Allow"><require from="Activity" to="RegistrationUserActivity"><var
        name="int␣Registration.idEvent"/></require></specification>
    <specification type="Allow"><require from="Activity" to="RegistrationUserActivity"><var
        name="int␣ActivityUser.idActivity"/></require></specification>
    <specification type="Allow"><require from="Activity" to="RegistrationUserActivity"><var
        name="int␣ActivityUser.idUser"/></require></specification>
    <specification type="Allow"><require from="User" to="RegistrationUserActivity"><var name="int␣
        Registration.idUser"/></require></specification
    ><specification type="Allow"><require from="User" to="RegistrationUserActivity"><var
        name="int␣ActivityUser.idUser"/></require></specification>
    <specification type="Allow"><require from="Registration" to="User"><var name="int␣
        Registration.idUser"/></require></specification>
    <specification type="Allow"><require from="Registration" to="User"><var name="int␣
        ActivityUser.idUser"/></require></specification>
    <specification type="Allow"><require from="User" to="Registration"><var name="int␣
        Registration.idUser"/></require></specification>
    <specification type="Allow"><require from="User" to="Registration"><var name="int␣
        ActivityUser.idUser"/></require></specification>
    <specification type="Allow"><require from="Registration" to="RegistrationUserActivity"><var
        name="Database␣RegistrationRepositoryNewBase.dataBase"/></require></specification>
    <specification type="Allow"><require from="Registration" to="RegistrationUserActivity"><var
        name="int␣Registration.idEvent"/></require></specification>
    <specification type="Allow"><require from="Registration" to="RegistrationUserActivity"><var
        name="int␣Registration.idUser"/></require></specification>
    <specification type="Allow"><require from="Registration" to="RegistrationUserActivity"><var
        name="float␣Registration.totalValue"/></require></specification>
    <specification type="Allow"><require from="Registration" to="RegistrationUserActivity"><var
        name="int␣ActivityUser.idUser"/></require></specification>
    <specification type="Allow"><require from="RegistrationUserActivity" to="Registration"><var
        name="Database_RegistrationRepositoryNewBase.dataBase"/></require></specification>
    <specification type="Allow"><require from="RegistrationUserActivity" to="Registration"><var
        name="int_Registration.idEvent"/></require></specification>
    <specification type="Allow"><require from="RegistrationUserActivity" to="Registration"><var
        name="int_Registration.idUser"/></require></specification>
    <specification type="Allow"><require from="RegistrationUserActivity" to="Registration"><var
        name="float_Registration.totalValue"/></require></specification>
    <specification type="Allow"><require from="RegistrationUserActivity" to="Registration"><var
        name="int_ActivityUser.idUser"/></require></specification>
    <specification type="Allow"><require from="Registration" to="RegistrationUserActivity"><var
        name="Payment$TypePayment_Payment.paymentType"/></require></specification>
    <specification type="Allow"><require from="Registration" to="RegistrationUserActivity"><var
        name="Payment$StatusPayment_Payment.status"/></require></specification>
    <specification type="Allow"><require from="Registration" to="RegistrationUserActivity"><var
        name="float_Payment.value"/></require></specification>
    <specification type="Allow"><require from="Registration" to="RegistrationUserActivity"><var
        name="String_Payment.barcode"/></require></specification>
    <specification type="Allow"><require from="Registration" to="RegistrationUserActivity"><var
        name="String_Payment.date"/></require></specification>
    <specification type="Allow"><require from="RegistrationUserActivity" to="Registration"><var
        name="Payment$TypePayment_Payment.paymentType"/></require></specification>
    <specification type="Allow"><require from="RegistrationUserActivity" to="Registration"><var
        name="Payment$StatusPayment_Payment.status"/></require></specification>
    <specification type="Allow"><require from="RegistrationUserActivity" to="Registration"><var
        name="float_Payment.value"/></require></specification>
    <specification type="Allow"><require from="RegistrationUserActivity" to="Registration"><var
        name="String_Payment.barcode"/></require></specification>
    <specification type="Allow"><require from="RegistrationUserActivity" to="Registration"><var
        name="String_Payment.date"/></require></specification>
    <specification type="Allow"><require from="RegistrationUserActivity" to="Registration"><var
        name="Exception_e"/></require></specification>
    <specification type="Allow"><require from="Payment" to="Registration"><var name="Database_
        PaymentRepositoryNewBase.dataBase"/></require></specification>
    <specification type="Allow"><require from="Payment" to="Registration"><var
        name="Payment$TypePayment_Payment.paymentType"/></require></specification>
    <specification type="Allow"><require from="Payment" to="Registration"><var
        name="Payment$StatusPayment_Payment.status"/></require></specification>
    <specification type="Allow"><require from="Payment" to="Registration"><var name="float_
        Payment.value"/></require></specification>
    <specification type="Allow"><require from="Payment" to="Registration"><var name="String_
        Payment.barcode"/></require></specification>
    <specification type="Allow"><require from="Payment" to="Registration"><var name="String_
        Payment.date"/></require></specification>
    <specification type="Allow"><require from="Payment" to="Registration"><var name="Exception_
        e"/></require></specification>
    <specification type="Allow"><require from="Registration" to="Payment"><var name="Database_
        PaymentRepositoryNewBase.dataBase"/></require></specification>
    <specification type="Allow"><require from="Registration" to="Payment"><var
        name="Payment$TypePayment_Payment.paymentType"/></require></specification>
    <specification type="Allow"><require from="Registration" to="Payment"><var
        name="Payment$StatusPayment_Payment.status"/></require></specification>
    <specification type="Allow"><require from="Registration" to="Payment"><var name="float_
        Payment.value"/></require></specification>
    <specification type="Allow"><require from="Registration" to="Payment"><var name="String_
        Payment.barcode"/></require></specification>
    <specification type="Allow"><require from="Registration" to="Payment"><var name="String_
        Payment.date"/></require></specification>
    <specification type="Allow"><require from="Registration" to="Payment"><var name="Exception_
        e"/></require></specification>
    <specification type="Allow"><require from="PaymentCash" to="Registration"/></specification>
    <specification type="Allow"><require from="PaymentCash" to="Payment"/></specification>
    <specification type="Allow"><require from="PaymentCash"
        to="RegistrationUserActivity"/></specification>
    <specification type="Allow"><require from="Payment" to="RegistrationUserActivity"><var
        name="Payment$TypePayment_Payment.paymentType"/></require></specification>
    <specification type="Allow"><require from="Payment" to="RegistrationUserActivity"><var
        name="Payment$StatusPayment_Payment.status"/></require></specification>
    <specification type="Allow"><require from="Payment" to="RegistrationUserActivity"><var
        name="float_Payment.value"/></require></specification>
    <specification type="Allow"><require from="Payment" to="RegistrationUserActivity"><var
        name="String_Payment.barcode"/></require></specification>
    <specification type="Allow"><require from="Payment" to="RegistrationUserActivity"><var
        name="String_Payment.date"/></require></specification>
    <specification type="Allow"><require from="RegistrationUserActivity" to="Payment"><var
        name="Payment$TypePayment_Payment.paymentType"/></require></specification>
    <specification type="Allow"><require from="RegistrationUserActivity" to="Payment"><var
        name="Payment$StatusPayment_Payment.status"/></require></specification>
    <specification type="Allow"><require from="RegistrationUserActivity" to="Payment"><var
        name="float_Payment.value"/></require></specification>
    <specification type="Allow"><require from="RegistrationUserActivity" to="Payment"><var
        name="String_Payment.barcode"/></require></specification>
    <specification type="Allow"><require from="RegistrationUserActivity" to="Payment"><var
        name="String_Payment.date"/></require></specification>
    <specification type="Allow"><require from="RegistrationUserActivity" to="Payment"><var
        name="Exception_e"/></require></specification>
    <specification type="Allow"><require from="Review" to="Reviewer"><var name="int_
        Assignment.idReviwerUser"/></require></specification>
    <specification type="Allow"><require from="Reviewer" to="Review"><var name="int_
        Assignment.idReviwerUser"/></require></specification>
    <specification type="Allow"><require from="Reviewer" to="Review"><var name="Exception␣e1"/></require></specification>
    <specification type="Allow"><require from="Assignment" to="CompleteSubmission"><var name="int␣Assignment.idReviewSubmission"/></require></specification>
    <specification type="Allow"><require from="CompleteSubmission" to="Assignment"><var name="int␣Assignment.idReviewSubmission"/></require></specification><specification type="Allow"><require from="Assignment" to="Reviewer"><var name="int␣Assignment.idReviwerUser"/></require></specification>
    <specification type="Allow"><require from="Reviewer" to="Assignment"><var name="int␣Assignment.idReviwerUser"/></require></specification>
    <specification type="Allow"><require from="Reviewer" to="Assignment"><var name="Exception␣e1"/></require></specification>
    <specification type="Allow"><require from="Assignment" to="Review"><var name="int␣Assignment.idReview"/></require></specification>
    <specification type="Allow"><require from="Assignment" to="Review"><var name="int␣Assignment.idReviwerUser"/></require></specification>
    <specification type="Allow"><require from="Review" to="Assignment"><var name="int␣Assignment.idReview"/></require></specification>
    <specification type="Allow"><require from="Review" to="Assignment"><var name="int␣Assignment.idReviwerUser"/></require></specification>
    <specification type="Allow"><require from="Review" to="Assignment"><var name="Exception␣e1"/></require></specification>
    <specification type="Forbid"><suppress from="Assignment" to="InsertAuthor"><var name="User␣user"/></suppress></specification>
    <specification type="Allow"><require from="User" to="CompleteSubmission"><var name="User␣user"/></require></specification>
    <specification type="Allow"><require from="CompleteSubmission" to="InsertAuthor"><var name="Author␣author"/></require></specification>
    <specification type="Allow"><require from="CompleteSubmission" to="User"><var name="User␣user"/></require></specification>
    <specification type="Allow"><require from="User" to="InsertAuthor"><var name="Exception␣e"/></require></specification>
    <specification type="Allow"><require from="User" to="Reviewer"><var name="Exception␣e"/></require></specification>
    <specification type="Allow"><require from="User" to="CompleteSubmission"><var name="Exception␣e"/></require></specification>
    <specification type="Allow"><require from="CompleteSubmission" to="Reviewer"><var name="Exception␣e"/></require></specification>
    <specification type="Allow"><require from="CompleteSubmission" to="InsertAuthor"><var name="String␣authorFiliation"/></require></specification>
    <specification type="Allow"><require from="CompleteSubmission" to="InsertAuthor"><var name="Exception␣e"/></require></specification
    ><specification type="Allow"><require from="ConflictOfInterest" to="CompleteSubmission"><var name="String␣authorFiliation"/></require></specification>
    <specification type="Allow"><require from="ConflictOfInterest" to="InsertAuthor"><var name="String␣authorFiliation"/></require></specification>
    <specification type="Allow"><require from="ConflictOfInterest" to="InsertAuthor"><var name="String␣reviewerFiliation"/></require></specification>
    <specification type="Allow"><require from="ConflictOfInterest" to="InsertAuthor"><var name="String␣userFiliation"/></require></specification>
    <specification type="Allow"><require from="InsertAuthor" to="ConflictOfInterest"><var name="String␣authorFiliation"/></require></specification>
    <specification type="Allow"><require from="InsertAuthor" to="ConflictOfInterest"><var name="String␣reviewerFiliation"/></require></specification>
    <specification type="Allow"><require from="InsertAuthor" to="ConflictOfInterest"><var name="String␣userFiliation"/></require></specification>
    <specification type="Allow"><require from="InsertAuthor" to="ConflictOfInterest"><var name="Exception␣e"/></require></specification>
    <specification type="Allow"><require from="User" to="ConflictOfInterest"><var name="Exception␣e"/></require></specification>
    <specification type="Allow"><require from="InsertAuthor" to="Reviewer"><var name="String␣reviewerFiliation"/></require></specification>
    <specification type="Allow"><require from="InsertAuthor" to="Reviewer"><var name="String␣userFiliation"/></require></specification>
    <specification type="Allow"><require from="Reviewer" to="InsertAuthor"><var name="String␣reviewerFiliation"/></require></specification>
    <specification type="Allow"><require from="Reviewer" to="InsertAuthor"><var name="String␣userFiliation"/></require></specification>
    <specification type="Allow"><require from="Reviewer" to="InsertAuthor"><var name="Exception␣e"/></require></specification>
    <specification type="Allow"><require from="InsertAuthor" to="CompleteSubmission"><var name="String␣authorFiliation"/></require></specification>
    <specification type="Allow"><require from="CompleteSubmission" to="ConflictOfInterest"><var name="String␣authorFiliation"/></require></specification>
    <specification type="Allow"><require from="CompleteSubmission" to="ConflictOfInterest"><var name="Exception␣e"/></require></specification>
    <specification type="Allow"><require from="Reviewer" to="ConflictOfInterest"><var name="String␣reviewerFiliation"/></require></specification>
    <specification type="Allow"><require from="Reviewer" to="ConflictOfInterest"><var name="String␣userFiliation"/></require></specification>
    <specification type="Allow"><require from="Reviewer" to="ConflictOfInterest"><var name="Exception␣e"/></require></specification>
    <specification type="Allow"><require from="ConflictOfInterest" to="Reviewer"><var name="String␣reviewerFiliation"/></require></specification>
    <specification type="Allow"><require from="ConflictOfInterest" to="Reviewer"><var name="String␣userFiliation"/></require></specification>
</system>
```

C.2 TEST SUITE GRAPHS

T1 Analysis

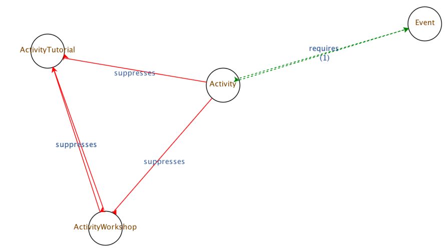


Figure 1. Complete Graph

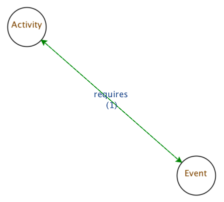


Figure 2. Graph generated after fixing the bug

Activity
Event

Figure 3. After judge the interactions

T2 Analysis

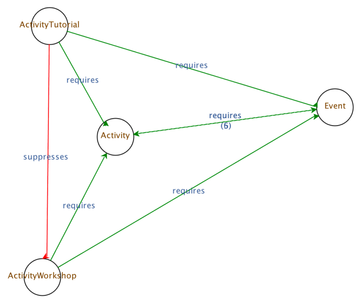


Figure 2. Complete Graph

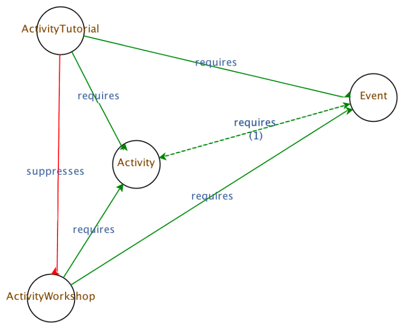


Figure 5. Reduced Graph used in the analysis

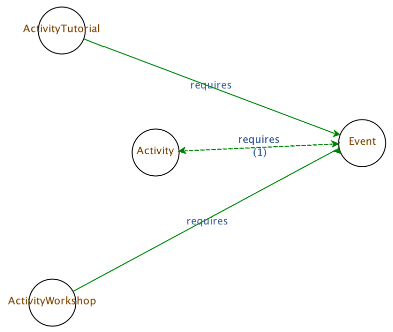


Figure 3. Graph generated after fixing the bug

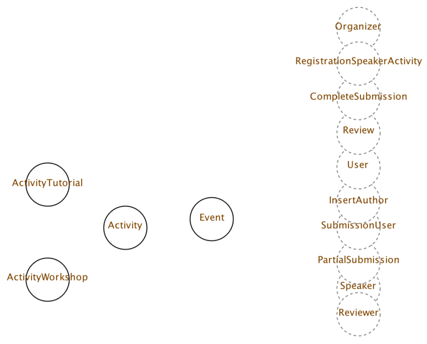


Figure 7. After judge the interactions

T3 Analysis

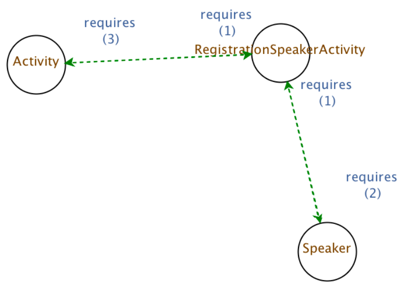


Figure 8. The complete and reduced graphs are the same to this test because none spec matched

Speaker
Activity
RegistrationSpeakerActivity

Figure 9. After judge all interactions

T4 Analysis

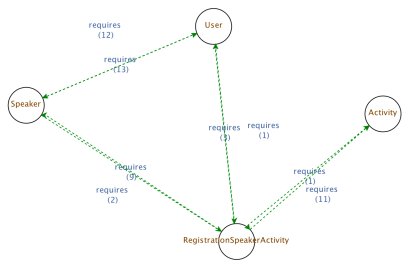


Figure 10. Complete Graph

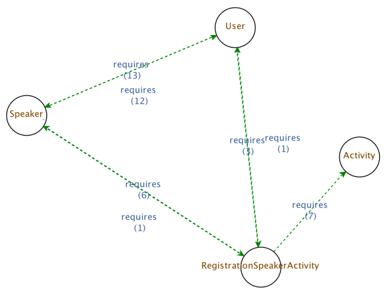


Figure 11. Reduced Graph used in the analysis

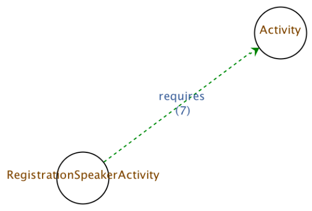


Figure 12. After judge all interactions

T5 Analysis

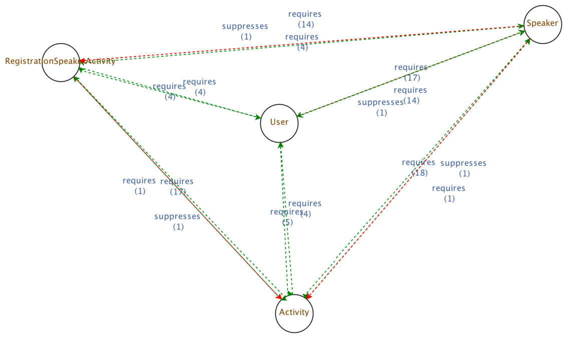


Figure 13. Complete Graph

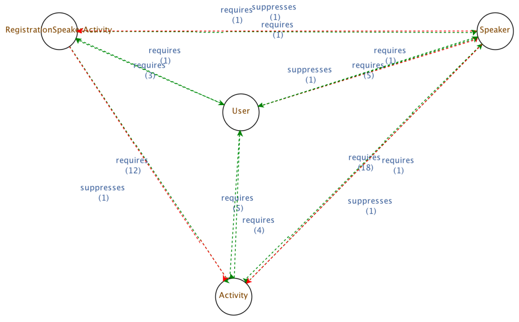


Figure 14. Reduced Graph used in the analysis

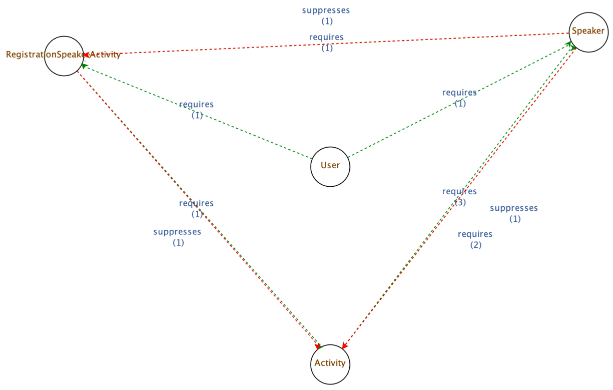


Figure 15. Graph generated after fixing the bug

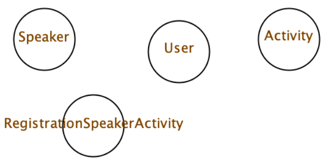


Figure 16. After judge all interactions

T6 Analysis

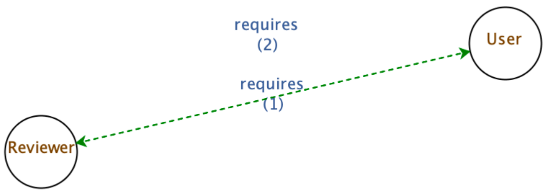


Figure 17. The complete and reduced graphs are the same to this test because none spec matched

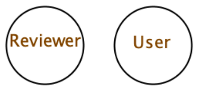


Figure 18. After judge all interactions

T7 Analysis

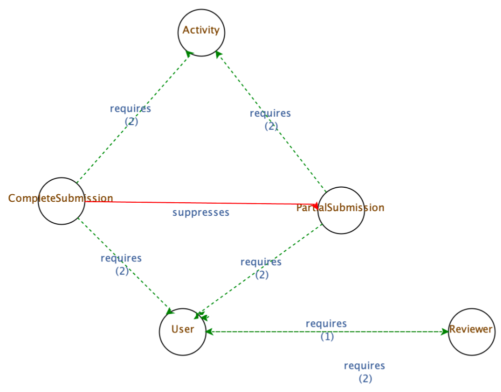


Figure 19. Complete Graph

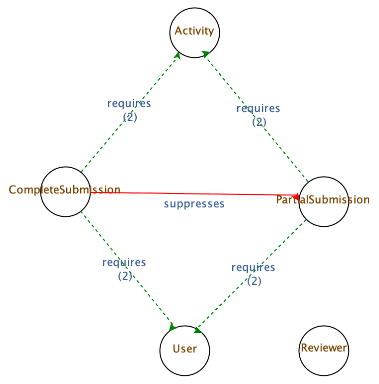


Figure 20. Reduced Graph used in the analysis

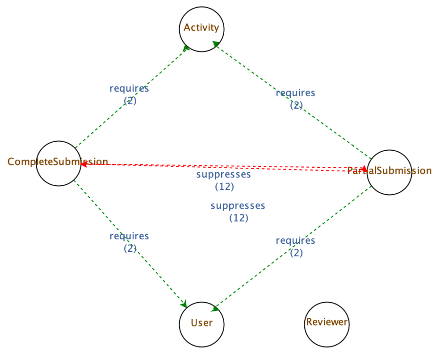


Figure 21. Graph generated after fixing the bug

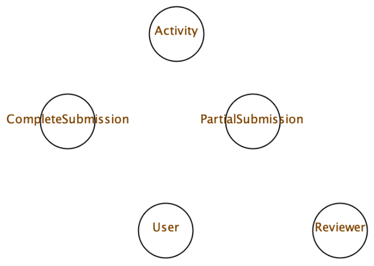


Figure 22. After judge all interactions

T8 Analysis

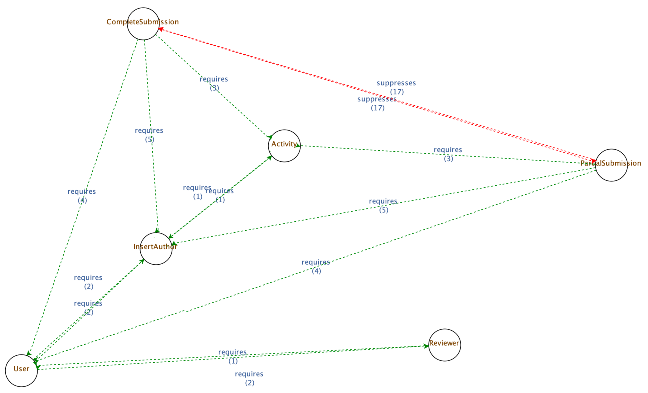


Figure 23. Complete Graph

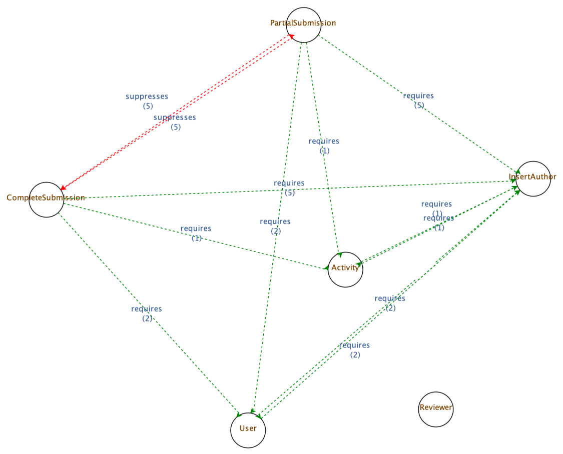


Figure 24. Reduced Graph used in the analysis

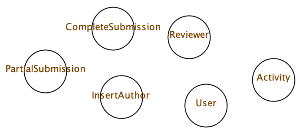


Figure 25. After judge all interactions

T9 Analysis

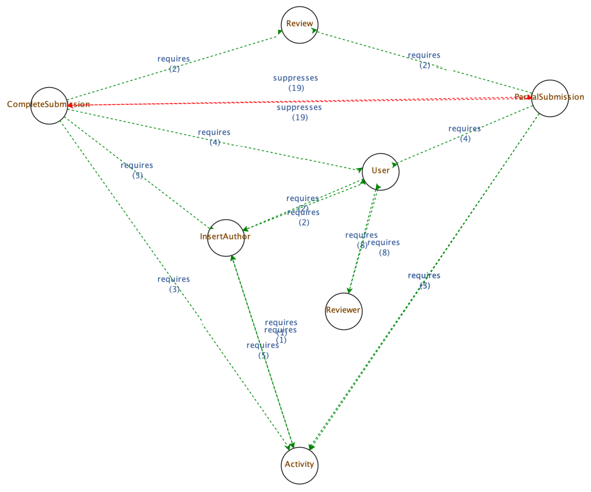


Figure 26. Complete Graph

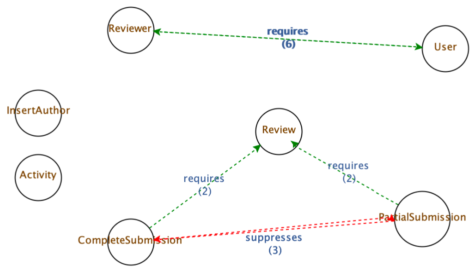


Figure 27. Reduced Graph used in the analysis

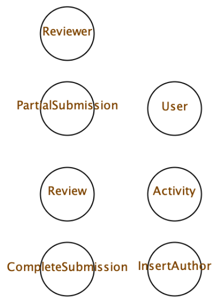


Figure 28. After judge all interactions

T10 Analysis

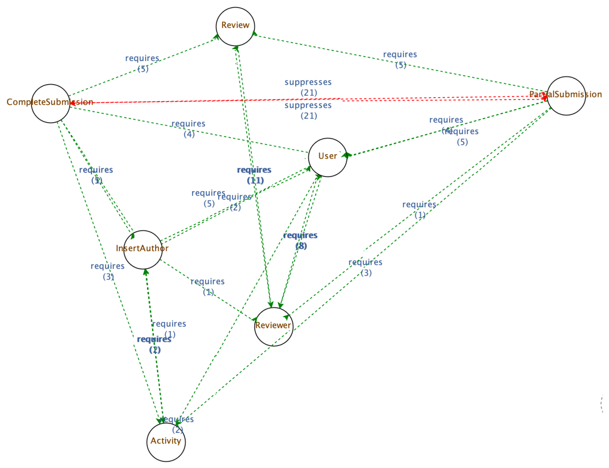


Figure 29. Complete Graph

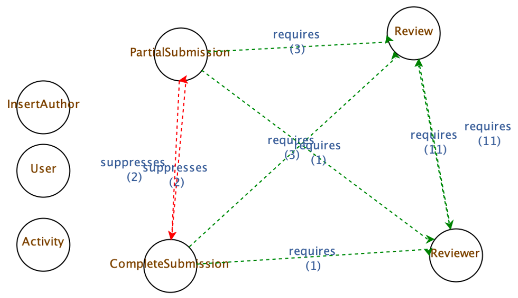


Figure 30. Reduced Graph used in the analysis

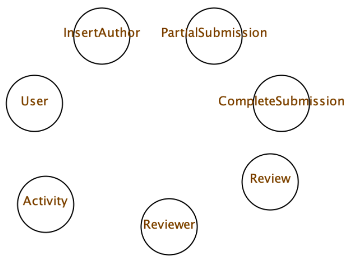


Figure 31. After judge the interactions

T11 Analysis

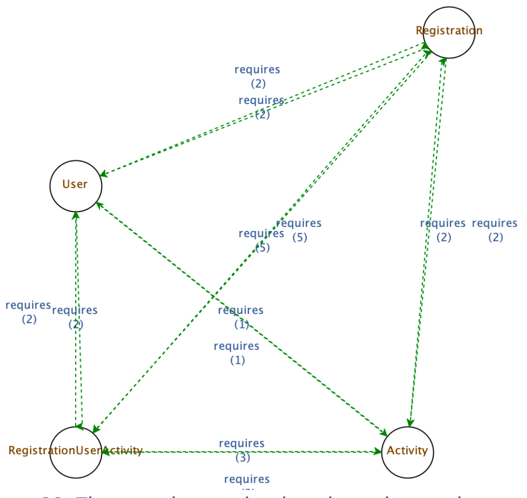


Figure 32. The complete and reduced graphs are the same to this test because none spec matched

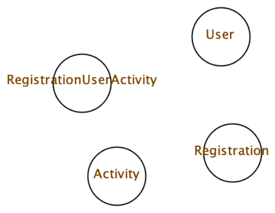


Figure 33. After judge the interactions

T12 Analysis

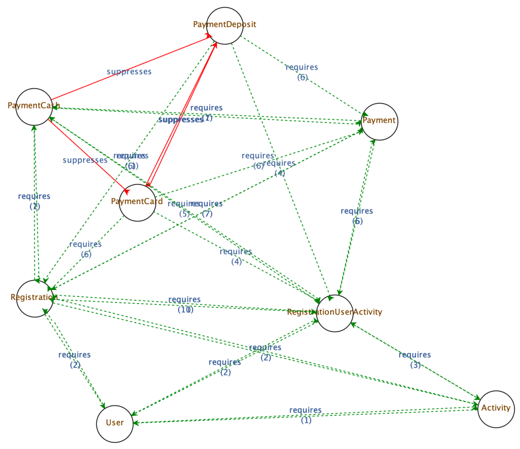


Figure 34. Complete Graph

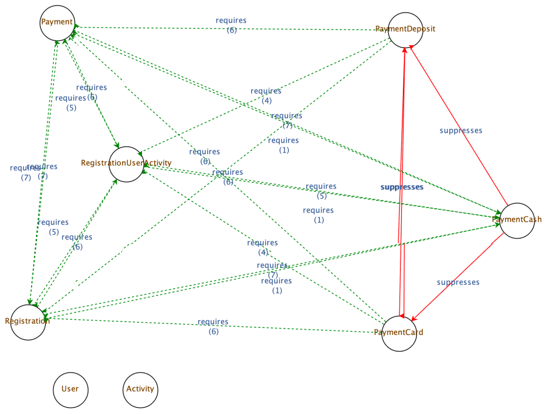


Figure 35. Reduced Graph used in the analysis

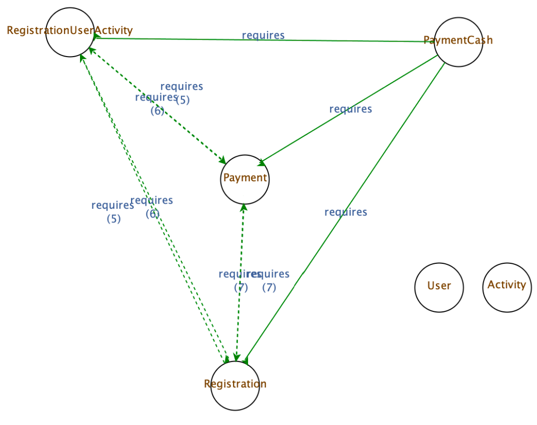


Figure 36. Graph generated after fixing the bug

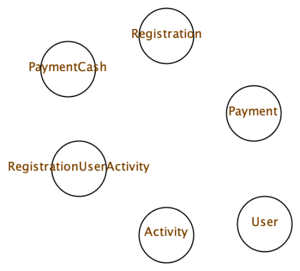


Figure 37. After judge the interactions

T13 Analysis

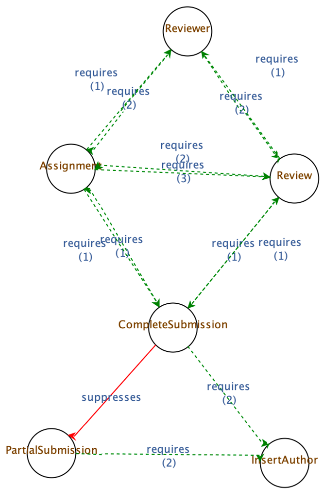


Figure 38. Complete Graph

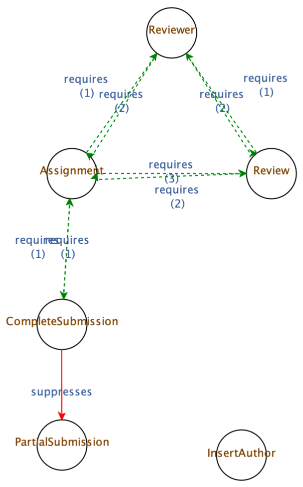


Figure 39. Reduced Graph used in the analysis

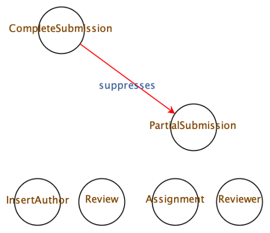


Figure 40. After judge the interactions

T14 Analysis

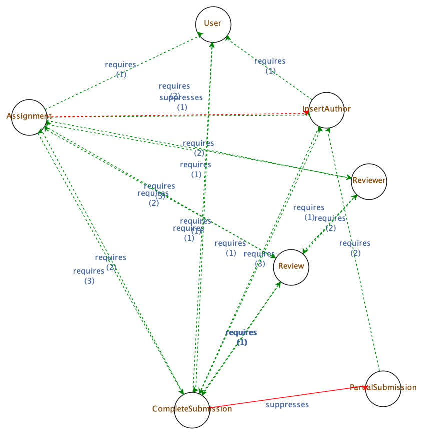


Figure 41. Complete Graph

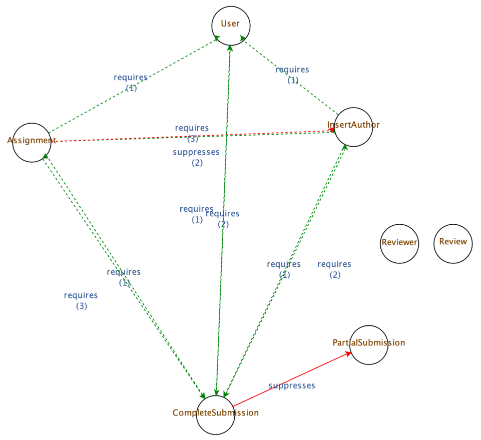


Figure 42. Reduced Graph used in the analysis

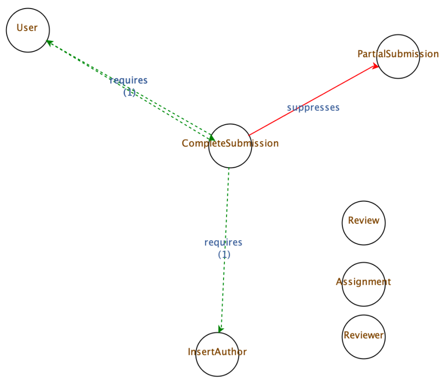


Figure 43. Graph generated after fixing the bug

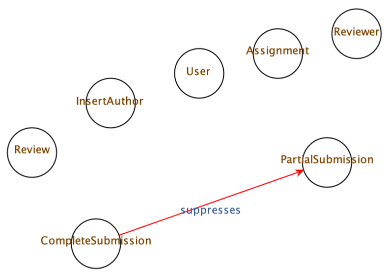


Figure 44. After judge the interactions

T15 Analysis

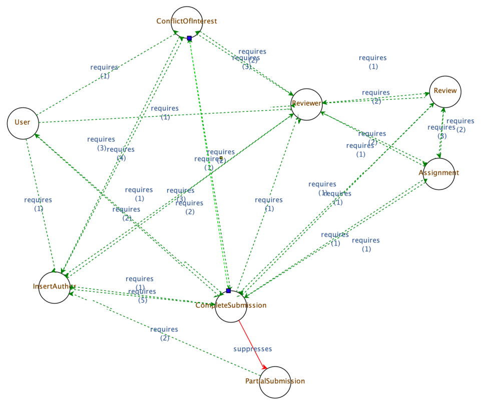


Figure 45. Complete Graph

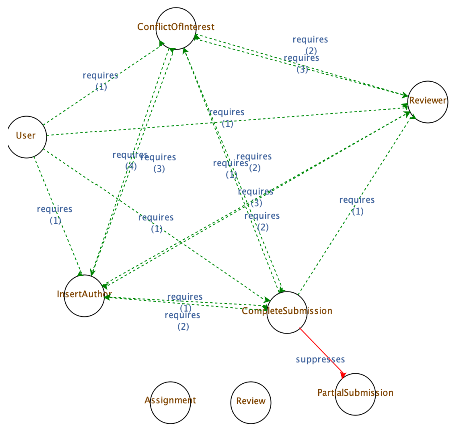


Figure 46. Reduced Graph used in the analysis

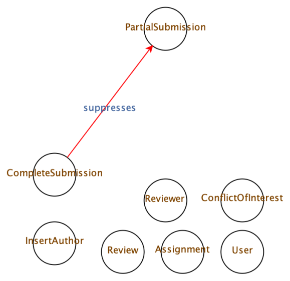


Figure 47. After judge the interactions

This volume has been typeset in LATEXwith the UFBAThesis class (¡www.dcc.ufba.br/˜flach/ufbathesis¿). For details about this document, click here.