CONFIGURAÇÃO DE PRODUTOS EM LINHAS DE PRODUTO DE SOFTWARE USANDO TÉCNICAS DE BUSCA

JULIANA ALVES PEREIRA

CONFIGURAÇÃO DE PRODUTOS EM LINHAS DE PRODUTO DE SOFTWARE USANDO TÉCNICAS DE BUSCA

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

Orientador: Eduardo Magno Lages Figueiredo
Coorientador: Thiago Ferreira de Noronha

Belo Horizonte
Abril de 2014

JULIANA ALVES PEREIRA

SEARCH-BASED PRODUCT CONFIGURATION IN SOFTWARE PRODUCT LINES

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Eduardo Magno Lages Figueiredo
Co-Advisor: Thiago Ferreira de Noronha

Belo Horizonte
April 2014

Pereira, Juliana Alves

P436s Search-Based Product Configuration in Software Product Lines / Juliana Alves Pereira. — Belo Horizonte, 2014
xxiv, 66 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais. Departamento de Ciência da Computação.

Orientador: Eduardo Magno Lages Figueiredo Coorientador: Thiago Ferreira de Noronha

1. Computação - Teses. 2. Engenharia de software - Teses. 3. Otimização - Teses. I. Orientador. II. Coorientador. III. Título.

CDU 519.6*32 (043)

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

FOLHA DE APROVAÇÃO

Search-based product configuration in software product lines

JULIANA ALVES PEREIRA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

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

PROF. THIAGO FERREIRA DE NORONHA - Coorientador
Departamento de Ciência da Computação - UFMG

PROF. ALESSANDRO FABRICIO GARCIA
Departamento de Informática - PUCRJ

PROF. SEBASTIÁN ALBERTO URRUTIA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 06 de maio de 2014.

I dedicate this achievement to God who always stood beside me.

Acknowledgments

This work is the result of the effort and learning provided to me and coming of several people who directly and indirectly contributed to this achievement.

First of all I am thankful to God for life. My family for their support and attention. To my parents, Ana Maria and Denisio, by love and dedication that taught me the values of life and provided me with an education that was the basis of my personal and professional formation. To my brothers, Fernando and Jaqueline, for the words of encouragement and the constant presence in my life. To my husband Johny thank you for your patience, the love, affection, and for all unconditional support at all times. To my brother in law Kaique, my mother in-law Mires and my father in-law Sebastiao by the precious moments of sharing, prayers and positive thinking.

I am lucky to have the opportunity for collaboration with two brilliant researchers who contributed a lot to this dissertation. First, I would like to specially thank my advisor Dr. Eduardo Figueiredo for the availability, collaboration, knowledge transmitted and ability to stimulate along this master dissertation. I am very grateful for the quality supervision I received, for your friendship, for moments of reflection and discussion that contributed to the enrichment of my learning during this work. I extend my gratitude to my co-advisor, Dr. Thiago Noronha, who has always actively participated in my work, discussions and meetings, always with great ideas and suggestions that contributed to the enrichment and results achieved in this research.

I am thankful to my friends, especially to lab friends LabSoft, by time divided together. To teachers of the Master in Computer Science at UFMG, thank you for the teachings and quality training I received in the course subjects. The secretariat of the DCC for all the support, attention and friendship. And, the Fapemig for financial support necessary to carry on this work.

To all those who collaborated directly or indirectly in this work my infinite gratitude for having believed in my abilities and especially by the support I received to achieve this dream.

Thank you!

"Victory is always possible for the person who refuses to stop fighting."
(Hill, Napoleon)

Resumo

Linha de produtos de software (LPS) é um conjunto de sistemas de software que compartilham um conjunto comum de características que satisfaçam as necessidades específicas de um determinado segmento de mercado. Uma característica representa um incremento na funcionalidade relevante para alguns stakeholders. LPSs geralmente usam um modelo de características para capturar e documentar as características comuns e variáveis. O principal desafio de usar modelos de características é derivar uma configuração de produto que satisfaça os requisitos do negócio e do cliente. Embora o suporte automatizado para a configuração do produto já foi investigado na literatura, os requisitos do cliente são geralmente negligenciados. Esta dissertação apresenta uma abordagem de engenharia de software baseada em busca para resolver o problema de encontrar a configuração de produto ótima que maximiza a satisfação do cliente. Após modelar a configuração do produto em LPS como um problema de otimização, esta dissertação propõe um algoritmo exato e uma heurística para resolver o problema de configuração do produto. Experimentos computacionais mostraram que o algoritmo exato pode encontrar a configuração do produto ótima para instâncias reais da literatura e que a diferença relativa entre o resultado heurístico e a solução ótima é de no máximo 3%.

Palavras-chave : Linhas de Produto de Software, Configuração do Produto, Otimização Combinatória, Engenharia de Software Baseada em Busca.

Abstract

Software product line (SPL) is a set of software systems that share a common set of features satisfying the specific needs of a particular market segment. A feature represents an increment in functionality relevant to some stakeholders. SPLs commonly use a feature model to capture and document common and varying features. The key challenge of using feature models is to derive a product configuration that satisfies business and customer requirements. Although automated support for product configuration has already been investigated in the literature, customer requirements are usually neglected. This dissertation presents a search-based software engineering approach to tackle the problem of finding the optimal product configuration that maximizes the customer satisfaction. After modeling SPL product configuration as an optimization problem, this dissertation proposes an exact algorithm and a heuristic to solve the product configuration problem. Computational experiments showed that the exact algorithm can find the optimal product configuration for real-life SPL instances found in the literature and that the relative optimality gap of the heuristic is at most 3%.

Keywords: Software Product Lines, Product Configuration, Combinatorial Optimization, Search-Based Software Engineering.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

The growing need for developing larger and more complex software systems demands better support for reusable software artifacts [Pohl et al., 2005]. In order to address these demands, software product line (SPL) has been increasingly adopted in software industry [Clements and Northrop, 2001; Deelstra et al., 2004; van der Linden et al., 2007; Apel et al., 2013]. SPL is a set of software systems that share a common and variable set of features satisfying the specific needs of a particular market segment [Pohl et al., 2005]. It is built around a set of common software components with points of variability that allow product configuration [Clements and Northrop, 2001; van der Linden et al., 2007]. Large companies, such as Hewlett-Packard, Nokia, Motorola, and Dell have adopted SPL practices [Apel et al., 2013].

An important concept of an SPL is the feature model. Feature models are used to represent the common and variable features in SPL [Czarnecki and Eisenecker, 2000; Kang et al., 1990]. A feature represents an increment in functionality or a system property relevant to some stakeholders [Batory, 2005; Kang et al., 1990]. It may refer to functional requirements [Jarzabek et al., 2003], architecture decisions [Bernardo et al., 2002] or design patterns [Prehofer, 2001]. The potential benefits of SPLs are achieved through a software architecture designed to increase reuse of features in several SPL products. In practice, developing an SPL involves modeling of features representing different viewpoints, sub-systems, or concerns of the software system [Batory, 2005; Beuche et al., 2004]. There are common features found on all products of the product line (known as mandatory features) and variable features that allow distinguishing between products in a product line (generally represented by optional or alternative features). Variable features define points of variation and their role is to permit the instantiation of different products by enabling or disabling specific SPL functionality.

1.1 Problem Description and Motivation

A fundamental challenge in SPL is the process of enabling and disabling features in a feature model for a new software product instantiation. This process is called product configuration [Pohl et al., 2005]. It provides the required flexibility for product differentiation and diversification. As the number of features increase in a feature model, so does the number of product options in an SPL [Benavides et al., 2005]. Consequently, a large number of features lead to SPLs with a large number of products. SPL can easily incorporate several thousand of feature combinations, due to the continuous evolution of SPL, resulting in a combinatorial explosion of variants [Loesch and Ploedereder, 2007]. In practice, businesses use only a subset of these combinations. However, these subsets are not able to meet the different needs of customers. To stay competitive in today's marketplace, a company must understand its customers' wants and needs and design processes to meet their expectations and requirements. Therefore, a means of satisfying different customers would consider all possible configurations and thus specifically meet the needs of a particular customer. For instance, an SPL where all features are optional can instantiate 2^n products where n is the number of features. Consequently, product developers should be able to evaluate many possibilities to select the features that appropriately meet customer requirements. Moreover, once a feature is selected, it must be verified to conform to the myriad constraints in the feature model, turning this process into a complex, time-consuming, and error-prone task. Industrial-sized feature models with hundreds or thousands of features make this process impractical. Guidance and automatic support are needed to increase business efficiency when dealing with many possible combinations in an SPL.

In this context, the need of effective algorithms to analyze feature models has attracted the attention of researchers and practitioners [Batory, 2005; Benavides et al., 2005; Bryant, 1986; Czarnecki and Wasowski, 2007; Mendonça et al., 2009; White et al., 2009]. Several studies have proposed the use of specific logic-based systems, such as Constraint Satisfaction Problem (CSP) [Benavides et al., 2005; White et al., 2009], Binary Decision Diagrams (BDD) [Bryant, 1986; Czarnecki and Wasowski, 2007], and SAT solvers [Batory, 2005; Mendonça et al., 2009], to address different challenges related to SPL. For instance, White et al. [2009] present a technique to transform a flawed product configuration into a CSP and then derive the minimal set of feature selection changes to fix an invalid configuration. Similarly, Mendonça et al. [2009] relies on SAT solvers and BDD to automatically analyze feature models and provide support for interactive product configuration. However, none of these approaches consider the customer preferences as differential during the product configuration.

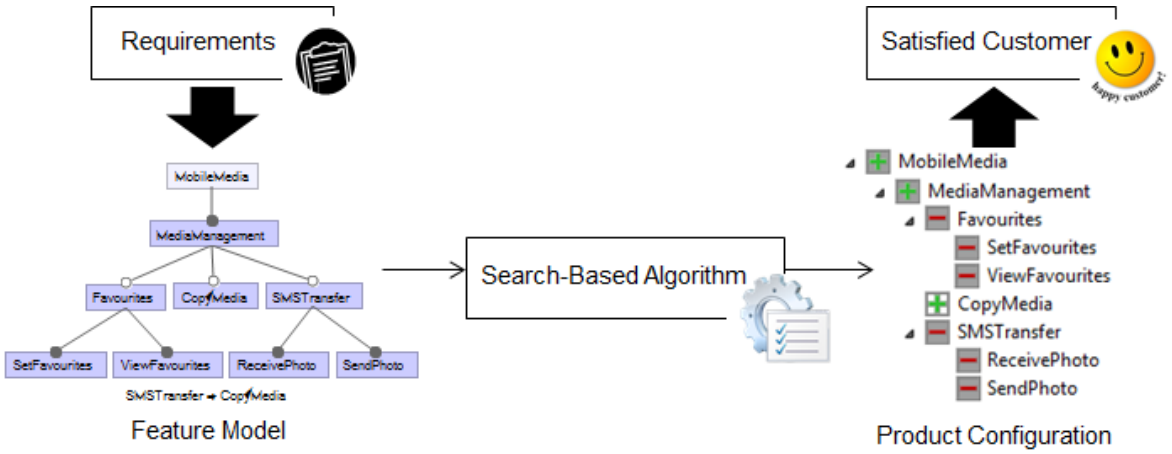

Figure 1.1. Overview of the Method to Automatic Product Configuration

1.2 Goal and Contributions

The main goal of this dissertation is to propose an automatic product configuration method based on search-based software engineering (SBSE) techniques. This method aims to maximize the customer satisfaction. Figure 1.1 presents an abstract overview of our method. As shown in Figure 1.1, from a feature model we use search-based algorithms to derive an optimized product configuration that maximizes the customer satisfaction, subject to business and customer requirements, and composition constraints (feature-tree and cross-tree constraints).

To achieve our goals, this dissertation presents four main contributions. The first contribution is a systematic literature review focusing on SPL management tools. The systematic review represents a significant step forward in the state-of-the-art by examining deeply many relevant tools for feature modeling. It contributes specifically with relevant information (i) to support practitioners choosing appropriate tools in a specific context of SPL, (ii) to check attributes and requirements relevant to those interested in developing new tools, and (iii) to the improvement/extension of existing tools. The second contribution of this dissertation is a comparative study of two tools for SPL variability management, which identifies common functionalities available. The results contributed with information to support extension of the product configuration functionality in one of the tools. The third contribution is the modeling of an optimization problem, an exact algorithm, and a heuristic algorithm in order to search for an optimized product configuration. Moreover, we evaluated the product configuration method by comparing the quality and scalability of the heuristic algorithm with the exact algorithm. The fourth contribution is a decision support system

for businesses. This system enables to reduce the effort required in task execution for product configuration. Ultimately, we expect the ideas discussed in this dissertation to raise awareness in our research field of the importance of the product configuration for supporting business models.

Parts of the results presented in this dissertation were published in the papers:

- Pereira, J., Figueiredo, E., and Noronha, T. F. (2013). Modelo Computacional para Apoiar a Configuração de Produtos em Linha de Produtos de Software. In: *Workshop de Engenharia de Software Baseada em Busca (WESB)*, co-located at CBSoft.

- Pereira, J., Souza, C. G., Figueiredo, E., Abilio, R., Vale, G., and Costa, H. (2013). Software Variability Management: An Exploratory Study with Two Feature Modeling Tools. In: *Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS)*, co-located at CBSoft.

- Pereira, J. and Figueiredo, E. (2013). Configuração de Produtos em Linha de Produtos de Software. In: *Workshop de Teses e Dissertações do CBSoft (WTD-Soft)*, co-located at CBSoft.

- Pereira, J.; Figueiredo, E., and Costa, H. (2012). Linha de Produtos de Software: Conceitos e Ferramentas. In: *VII Escola Regional de Informática de Minas Gerais (ERI-MG)*, co-located at SMC.

1.3 Dissertation Outline

This dissertation is organized as follows. Chapter 2 introduces background concepts about software product line (SPL) and search-based software engineering (SBSE). Chapter 3 first presents a systematic literature review focusing in SPL management tools, where it is possible to see which functionalities have been emphasized in past research and thus to identify gaps and opportunities for future research. Moreover, it presents a comparative study detailed with two SPL management tools. Finally, this chapter presents SPLConfig, a decision support system to automatic product configuration. Chapter 4 models the problem of SPL product configuration as an optimization problem and proposes two search-based algorithms to solve this problem. Chapter 5 presents and discusses the results of the computational experiments to evaluate the performance of the proposed algorithms in Chapter 4. Finally, Chapter 6 concludes this dissertation, discusses the limitations of related work, and proposes future research directions.

Chapter 2

Background

Software product line (SPL) is a set of software systems that share a common and variable set of features satisfying the specific needs of a particular market segment [Pohl et al., 2005]. It consists of two development processes, namely: domain engineering and application engineering [Apel et al., 2013]. In this chapter, we provide the background information necessary for reading this dissertation including domain engineering, application engineering, and search-based software engineering (SBSE). Section 2.1 defines domain engineering and shows how it can be represented. Section 2.2 defines application engineering and shows how it can be used for the construction of a unique and valid product configuration in SPLs. Section 2.3 describes SBSE and its importance in the process of application engineering. Finally, concluding remarks are discussed in Section 2.4.

2.1 Domain Engineering

Domain engineering is responsible for defining the commonality and variability of the SPL [Czarnecki et al., 2006; Pohl et al., 2005]. Commonality and variability are key concepts in SPL which are represented by feature models [Kang et al., 2002]. In a feature model, common features to the domain composes the core and others features composes the variation points [Pohl et al., 2005]. Over the past years, several features modeling techniques have been developed in order to document and manage variability [Chen and Babar, 2011; Sinnema and Deelstra, 2007]. The first feature model was proposed as part of the feature-oriented domain analysis (FODA) method [Kang et al., 1990]. It has been applied in a number of domains including mobile phones [Figueiredo et al., 2008], telecom systems [Griss et al., 1998], network protocols [Barbeau and Bordeleau, 2002], Linux kernel [Lotufo et al., 2010], among others.

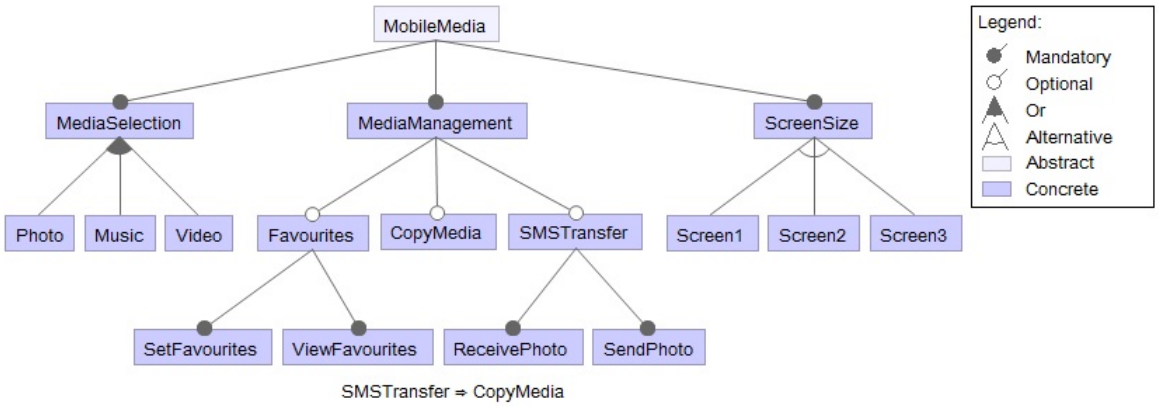

Figure 2.1. Example of a Feature Model for a Mobile Phone Product Line

Feature models represent the common and variable features in SPL using a feature tree [Kang et al., 2002]. In feature models, nodes represent features and edges show relationships between parent and child features [Batory, 2005; Czarnecki and Wasowski, 2007]. These relationships define as the features can be combined. As an example, the mobile phone industry uses features to specify and build software for configurable phones. The software product that goes into a phone is determined by the phone's features. Figure 2.1 depicts a simplified feature model of an SPL, called Mobile Media [Figueiredo et al., 2008], inspired by the mobile phone industry. Mobile Media is an SPL for applications that manipulate photo, music, and video on mobile devices, such as mobile phones. The optional and alternative features are configurable on selected mobile phones depending on the API support they provide. Mobile Media was developed for a family of 4 brands of devices, namely Nokia, Motorola, Siemens, and RIM.

Figure 2.1 illustrates the common graphical notation employed in feature modeling. A single root node, *MobileMedia*, represents the concept of the domain being modeled. By convention, we always assume that the root feature is part of all valid product configurations. Mandatory features are represented by filled circles, such as *MediaSelection*. Mandatory features must be selected in a product configuration if its parent feature is selected. Optional features are represented by empty circles, such as *Favourites*. Optional features can only be selected in a product configuration if its parent feature is selected. Alternatively, features can be grouped together into OR-groups or XOR-groups. Features that are part of such groups are called alternative features. XOR-groups are represented by interlinked edges and connected by an empty arc, such as *Screen1*, *Screen2* and *Screen3*. In XOR-groups only one feature can be selected

whenever the group's parent feature is selected. On the other hand, OR-groups are represented by interlinked edges and connected by a filled arc, such as *Photo*, *Music* and *Video*. In OR-groups at least one feature in the group must be selected whenever the group's parent feature is selected.

In addition to features and their relationships, feature models often need to contain additional composition rules [Czarnecki and Eisenecker, 2000]. Additional composition rules refer to additional cross-tree constraints to restrict feature combinations [Czarnecki et al., 2006]. Figure 2.1 presents a cross-tree constraint which indicates that the feature *SMSTransfer* requires the feature *CopyMedia* (*SMSTransfer* $\rightarrow$ *CopyMedia*). That is, in order to receive a photo via SMS, this photo has to be copied into an album. Cross-tree constraints are responsible for validating a combination of not connected features, i.e., they add new relations to the feature model not described in the feature tree. A cross-tree constraint can be written using the binary operators $\wedge$ (conjunction), $\vee$ (disjunction), $\rightarrow$ (implication), $\leftrightarrow$ (biconditional), the unary operator $\neg$ (negation), in addition to Boolean values and variables.

2.2 Application Engineering

Once an SPL is defined, application engineering refers to the ability of a product to be configured, customized, extended, or changed for use in a specific context [Goedicke et al., 2004; Pohl et al., 2005]. Feature model represents the space of all valid product configurations in an SPL [Czarnecki et al., 2006]. It illustrates specific points where a decision has to be made for product configuration according to some specific criteria, such as customer requirements (non-functional requirements) [Goedicke et al., 2004]. Customer requirements are particular characteristics and specifications of a good or service as determined by a customer.

The relationships between the features indicate the choices that can be made for product configuration. For example, in Figure 2.1, the alternative group *Screen1*, *Screen2* and *Screen3* indicates that developers must choose only one feature for implementing the screen size (i.e., the feature *ScreenSize*) in Mobile Media SPL. The Mobile Media SPL illustrated in Figure 2.1 is capable of configuring 126 different mobile phone products. This is possible through of the selection or exclusion of variables features in an SPL [Loesch and Ploedereder, 2007]. Figure 2.2 illustrates a sample product configuration for a mobile phone. This application has the basic features required for operation in the device: *MediaSelection* of type *Music*, *MediaManagement*, *CopyMedia,* and *ScreenSize* of type *Screen1*.

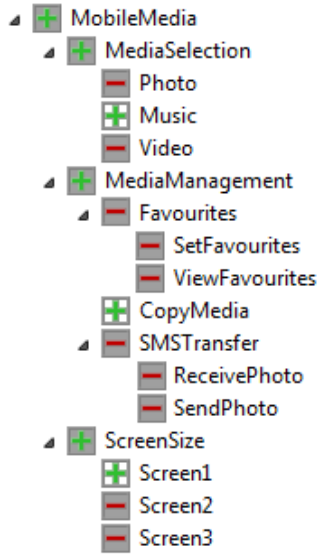

Figure 2.2. Example of a Sample Product Configuration

Some researchers have developed visualization techniques to assist developers in decision making during the product configuration process [Botterweck et al., 2007; Mendonça et al., 2009; Thüm et al., 2014]. However, industrial case studies have shown that product configuration is still a time-consuming and expensive activity [Deelstra et al., 2004]. For instance, during product configuration a feature model is configured manually through successive steps, by one or more developers, until a final product is obtained and provided to the customer. The manual processes commonly used to product configuration scale poorly for NP-hard problems. Moreover, these approaches focus more on functional requirements of a product and their dependencies and less on non-functional requirements.

Generally, companies sell small subsets of configurations. However, in practice these subsets are not able to meet the different need of customers. For example, in most cases customers pay for functionality that will never be used. Therefore, a key need with SPL is determining how to configure a set of features for a customer requirement set. In Figure 2.1, every feature refers to functional features of the Mobile Media SPL. However, every feature may have associated non-functional features [Kang et al., 1990]. For instance, considering the Mobile Media SPL, it is possible to identify non-functional features related to each feature, such as cost and level of importance (benefit) of each feature for the customer. It means that every product not only differs because of its functional features, but because of its non-functional features too. Therefore, we propose to extend feature models with non-functional features and we adapted in Figure 2.3 the proposed notation in Benavides et al. [2005] to our problem.

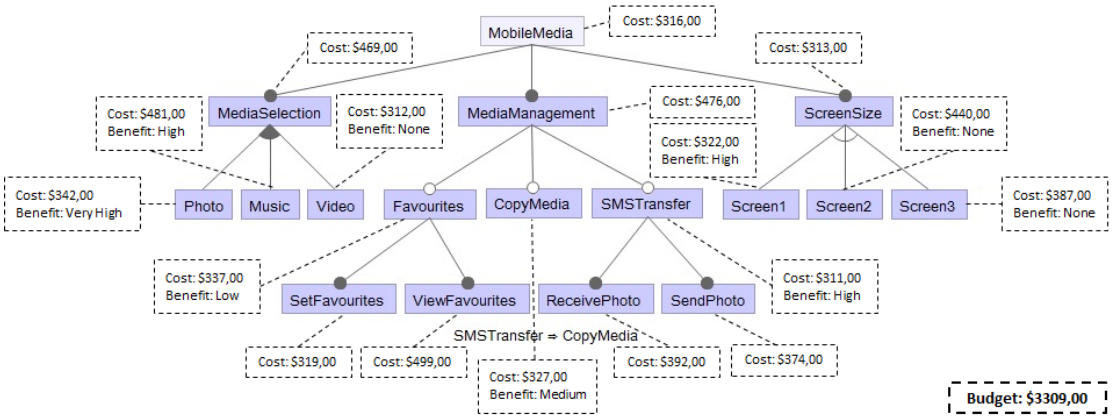

Figure 2.3. Example of a Decorate Feature Model with Non-Functional Features

Figure 2.3 illustrates the non-functional features in feature model of Figure 2.1. In this work we have considered only three non-functional requirements: cost, benefit and budget. In Figure 2.3, all features (mandatory and optional features) have the cost attribute, and only optional features have the benefit attribute. Mandatory features do not have benefit attribute because they must always be selected if its parent are selected, independent of its benefit. For example, the optional feature *Favourites* has cost and benefit attributes, while the mandatory feature *MediaSelection* has only the cost attribute. Note that the benefit attribute is classified into five levels: none, low, medium, high and very high. From meetings with the customers, the developers identify and describe the prioritization of features. Next, we use a conversion scheme for the benefit that ranges from 0% to 100%. Our goal is to find an optimal solution by means of an objective function that maximizes customer satisfaction (benefit/cost) without exceeding the available budget (in the lower right corner of the Figure 2.3). As a motivating example, given a mobile phones system that includes a variety of potential phones types, what is the phone that best meets the customer requirements and can be constructed with a given budget? The challenge is that with hundreds or thousands of features, it is hard to analyze the different product configurations to find an optimal configuration.

2.3 Search-Based Software Engineering

Search-based software engineering (SBSE) is the name given to a field of research in which optimization techniques are used to address problems in software engineering [Harman and Jones, 2001]. In this case, the term search is used to refer the search-based optimization algorithms that are used [Harman et al., 2012]. SBSE seeks

to reformulate software engineering problems as search-based optimization problems. Therefore, a search problem is one in which optimal or near-optimal solutions are sought in a search space of candidate solutions, guided by a fitness function that distinguishes between better and worse solutions [Harman et al., 2012].

SBSE has proved to be an applicable and successful field, with many studies across the software engineering life cycle, from requirements and project planning to maintenance and reengineering [Harman et al., 2012]. There is a repository of publications on SBSE available on the Web.[1] This repository includes over one thousand relevant publications from 1976 to 2014, where a wide variety of different optimization and search techniques have been used. For example, Local Search, Simulated Annealing (SA), and Genetic Algorithms (GAs). There is also increasing evidence of industrial interest in SBSE by many software-centric organizations including IBM [Yoo et al., 2011], Microsoft [Lakhotia et al., 2010], Motorola [Baker et al., 2006], and Nokia [Del Rosso, 2006].

A general trend towards SPL has taken to an increase for using of SBSE techniques. SBSE is relevant for SPLs because SBSE offers a suite of adaptive, automated and semiautomatic solutions in situations typified by large complex problem spaces with multiple competing and conflicting objectives. Basically, it has been used where there is a large set of choices and finding good solutions can be hard. An extensive number of SBSE studies for solving problems in SPL have been documented in the literature, such as the applicability of SBSE for automatic analysis of feature models. These studies use Constraint Satisfaction Problem (CSP) [Benavides et al., 2005; White et al., 2009], Binary Decision Diagrams (BDD) [Bryant, 1986; Czarnecki and Wasowski, 2007; Mendonça et al., 2009], SAT solvers [Batory, 2005; Mendonça et al., 2009], Alloy [Gheyi et al., 2006], Prolog [Beuche et al., 2004]. They work with important metrics associated with feature models that can be used to measure different aspects of the corresponding SPL. For example, the number of product configurations can be computed by counting the number of valid configurations in the feature model. Despite the relative success of methods that make use of search-based algorithms to address SPL-specific problems, there are still open issues related to optimization of the product configuration. Developers face a number of challenges when attempting to derive an optimized product configuration that satisfies a set of requirements. Therefore, this dissertation relies on SBSE techniques for automatic product configuration in SPL. Our study complements and extends manual techniques of product configuration.

[1] http://crestweb.cs.ucl.ac.uk/resources/sbse_repository/

2.4 Concluding Remarks

In this chapter, we introduced domain engineering and application engineering in SPLs. We showed how feature models can be used to build valid product configuration in SPL. In addition, we show that to quickly derive a product configuration that meets the customer requirements, developers need algorithmic techniques to automatically generate a feature selection that optimizes desired product properties. Finally, we discussed the use of search-based software engineering (SBSE) for solving problems in SPL, and argued that these techniques can be applied effectively in the application engineering during product configuration. The next chapter contributes specifically with a systematic literature review and a comparative study of tools that support SPL variability management. Our goal in the next chapter is identify a tool to add the functionality of automatic product configuration using SBSE techniques.

Chapter 3

SPL Management Tools

Since feature models are undergoing a rapid process of maturation, feature modeling tools are constantly being developed in practice. Despite the excellent SPL management tools, there remains, to date, no comprehensive survey of the whole field of tools. It is therefore timely to review the SPL tool literature, the relationships between the applications to which it has been applied, the techniques used, trends, and open problems. Section 3.1 extends the discussion on SPL by examining and classifying several tools to support SPL variability management discussed in the literature. It presents a survey of the state-of-the-art that identifies which functionalities have been emphasized in past research and thus to identify gaps and possibilities for future research. Section 3.2 describes a comparative study with two SPL variability management tools. Section 3.3 presents SPLConfig a tool to support SPL product configuration using search-based software engineering (SBSE) techniques. Finally, concluding remarks are discussed in Section 3.4.

3.1 Literature Review

Variability management tools are necessary to support the organizations in the SPL management. They provides the companies a guide for the development of SPLs, as well as a complete environment of development and maintenance of the SPL. However, the choose of one tool that best meets the companies SPL development goals is far from trivial. This is a critical activity, due to a sharp increase in the number of SPL management tools made available. Furthermore, tool support should assist the complete process, and not just some functionality, because it would lead to the need to use several tools and information traceability among them. In this context, this study contributes with a systematic literature review of tools to support SPL variability

management. The systematic review was conducted in order to identify, gather and classify tools in the literature that support the SPL management, including stages from conception until products derivation and SPL evolution.

Systematic review is one study method that has obtained much attention lately in software engineering [Kitchenham et al., 2009]. Our systematic review represents a significant step forward in the state-of-the-art by examining deeply many relevant tools for feature modeling. General propose of this study is to give a visual summary, by categorizing of existing tools to provide a search in journals and conference proceedings since 2000. Therefore, we have used a systematic and rigorous method to accomplish an extension study, identifying and selecting the reviewed primary studies. The next sections are organized as follows. Section 3.1.1 presents the steps carried by the systematic review. Section 3.1.2 reports and analyzes the results of this systematic review, and contributes specifically with relevant information to the extension of existing tools. Finally, Section 3.1.3 presents the threats to validity related for this systematic literature review and how they were addressed prior of the study to minimize their impact.

3.1.1 Study Settings

This study has been carried out according to the guideline for systematic literature review described in Kitchenham et al. [2009]. The guideline is structured according three-step process for *Planning, Conducting* and *Reporting* the review. Figure 3.1 depicts an overview of our research process comprising each step and its stages sequence. The execution of the overall process involves iteration, feedback, and refinement of the defined process [Kitchenham et al., 2009]. The systematic review steps, together with the protocol are detailed as follows.

Step 1. Has the goal of developing a protocol that specifies the plan that the systematic review will follow to identify, assess, and collate evidence [Kitchenham et al., 2009]. The *Planning* step includes several actions:

Identification of the need for a review. The need for a systematic review originates from increase in the number of SPL management tools made available. In this context, the choice of one tool that best fits practitioners needs in a specific context of SPL is far from trivial. Therefore, a systematic review aims to give a complete, comprehensive and valid picture of the tools available in the literature, in order to find out how the available tools are providing support to the process.

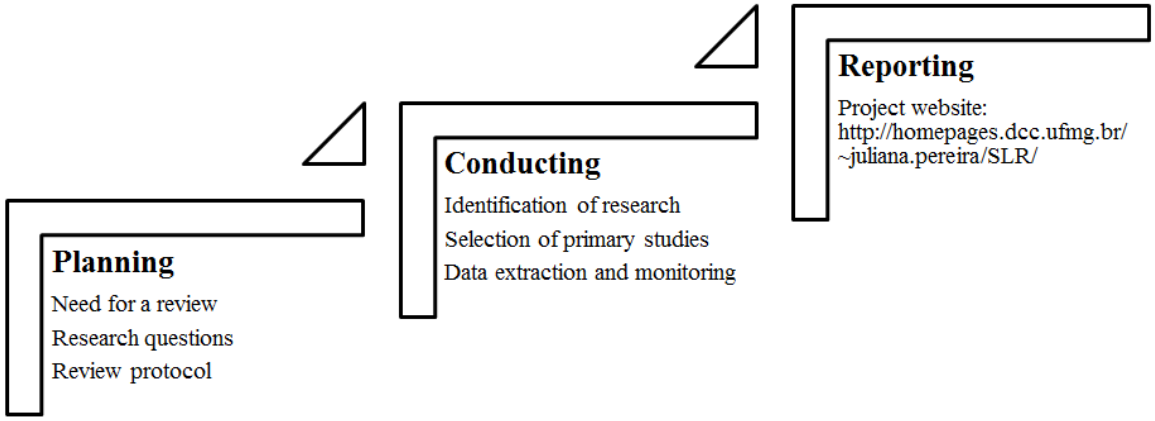

Figure 3.1. Overview Research Process

Specifying the research questions. The focuses of the research questions are identify SPL management tools, focusing since the SPL conception, to its development, maintenance and evolution. More specifically, we investigate three research questions (RQs) in this paper:

- *RQ1.* How many SPL management tools have been cited in the literature since 2000?
- *RQ2.* What are the main characteristics of the tools?
- *RQ3.* What are the main functionalities of the tools?

To address *RQ1*, we identified the tools that are being cited in the literature since 2000. With respect to *RQ2*, we identify the types of tools that are been developments and where the tools were developed. Through these descriptions, it is possible to map the current adoption of the tools. Finally, with respect to *RQ3*, we concerned with how the tool supports each stage of the development process, namely: domain engineering and application engineering. In particular, what SPL topics, contributions and novelty they constitute. Thus, it is possible to map how tools are supporting an SPL management process and if the process is not fully supported, i.e. there are many gaps in the existing tools, or if there is a necessity of developing functionalities that are not addressed by the tools. The analysis did not focus on the strengths or on the weaknesses of the tools, because the goal was to identify what the tools do and compare them. Therefore, the systematic review is conducted to identify, to analyze and interpret all available evidence related the research questions.

Developing a review protocol. We conducted a systematic literature review in journals and conferences proceedings published from January 1st 2000 to December 31th 2013, by the fact that visibility provided by SPL in recent years has produced a higher concentration of research [Lisboa et al., 2010]. Three researchers were involved in this process and all of them continuously discussed and refined the research questions, search strings, inclusion and exclusion criteria.

Step 2. *Conducting* the review means executing the protocol planned in the previous phase. To conduct a systematic literature review includes several actions:

Identification of research. Based on the research questions, some keywords were extracted and used to search the primary study sources. The search string used was constructed using the strategy by [Chen and Babar, 2011]. Following this strategy, search string addressed by this study is:

- *("management tool") AND ("feature modeling" OR "product configuration" OR "variability management") AND ("product line" OR "product family" OR "system family").*

The primary studies were identified by using a search string on three scientific databases: ACM Digital Library[1], IEEE Xplore[2] and ScienceDirect[3]. These libraries were chosen because they are some of the most relevant sources in software engineering [Travassos and Biolchini, 2007]. The search was performed using the specific syntax of each database and considering only the title, keywords and abstract. In addtion to the search in digital libraries, the references of the primary studies were also read in order to identify others primary studies relevant (technique called "snowballing").

Selection of primary studies. The basis for the selection of primary studies is the inclusion and exclusion criteria [Kitchenham et al., 2009]. We defined two inclusion criteria (IC):

- *IC1.* The publications should be "journal" or "conference" and only works written in English were considered.
- *IC2.* We included only primary studies that present tools to manage one or more phases of the development cycle and maintenance of SPLs. Therefore, the

[1]http://dl.acm.org/
[2]http://ieeexplore.ieee.org/
[3]http://www.sciencedirect.com/

abstract should explicitly mentions that the focus of the paper contributes with tools to support SPL variability management.

We defined six exclusion criteria (EC):

- *EC1.* We exclude technical reports presenting lessons learned, theses/dissertations, and duplicate papers. If a primary study is published in more than one paper, for example, if a conference paper is extended to a journal version only one instance should be counted as a primary study. Mostly, the journal version is preferred, as it is most complete.
- *EC2.* Studies that describe events, studies that are indexes or programming.
- *EC3.* Papers that not focus on SPL management tools. Approaches, methods, and techniques SPL by itself should be excluded.

After papers inclusion, during the selection tools, we apply the exclusion criteria the following:

- *EC4.* Tools that do not address the phases of development and maintenance of SPLs.
- *EC5.* We exclude tools that the project is currently discontinued.
- *EC6.* Tools without executable and/or documentation describing its functionalities available. Moreover, the tools with written documentation that does not have usable description about its functionalities were excluded, because it is not possible to describe how the functionality of the tools works.

Based on these criteria, each paper was included or not included by three different researchers (i.e. Pereira, Constantino and Figueiredo). Pereira coordinated the allocation of researchers to tasks based on the availability of each researcher and their ability to access the specific journals and conference proceedings. The researcher responsible for searching the specific journal or conference applied the detailed inclusion and exclusion criteria to the relevant papers. In addition, another researcher checked any papers included and excluded at this stage. This was done in order to check that all relevant papers were selected.

Data extraction and monitoring. For our study, we followed a systematic process shown in Figure 3.1. First, the reviewers read abstracts and looks for keywords and

concepts that reflect the contribution of the paper. While doing so the reviewer also identifies the context of the research. When abstracts are not enough, reviewers also read the introduction and conclusion sections. When the reviewers entered the data of a paper not relevant, they provided a short rationale why paper should not be included in the study (i.e, because SPL management tool not part of the contributions of the paper).

Once the list of primary studies is decided, the data from the tools cited by the papers is extracted. The phase of data extraction aims to summarize the data from the selected studies for further analysis. All documentation tools served as data sources for the data extraction, i.e. tutorials, technical reports, theses, websites, as well as the communication with authors. However, during the data extraction the exclusion criteria (*EC4*, *EC5* and *EC6*) were verified. The data extraction is designed based on the research questions. The data extracted from each tool selected were: (i) date of data extraction, (ii) primary studies (reference(s)), (iii) tool name, (iv) main reference, (iii) release year, (iv) website tool (if available), (v) main tools characteristics, (vii) where the tool was developed (academia or industry), and (vi) main functionalities each tool supports. The check the extraction involved all the authors of this paper. The data extraction and monitoring is detailed in Section 3.1.2.

Step 3. *Reporting* step follows to publish of the detailed results in the project website[4], in order to relating the review steps to the community. Therefore, it is clear to others how the search was, and how they can find the same documents. The website also provides more detailed information about the results of the search protocol (with complete lists of included and excluded primary studies) and the chosen tools.

3.1.2 Results and Analysis

Table 3.1 summarizes the number of papers of our systematic review. Note that for this analysis relevant studies (total included) more than tripled after 2009. In the first stage, after applied inclusion and exclusion criteria, 33 papers were included and 72 papers were excluded. In the second stage, we included 4 papers [Asikainen et al., 2004; Dhungana et al., 2007; Simmonds et al., 2011; Unphon, 2008] that also match our inclusion criteria. These papers were mined from references of included papers (technique called "snowballing"). This technique was necessary in order to have a more complete set of information and references about tools. At the end of the search, 37 papers were included for extracting and analyzing the data.

[4]http://homepages.dcc.ufmg.br/~juliana.pereira/SLR/

Table 3.1. Number Searched for Years 2000-2013

Year	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	All
Total	1	1	4	2	1	2	7	7	7	10	16	16	15	16	105
Total Selected	0	0	0	1	0	1	2	2	2	5	6	6	4	4	33
Total Snowballing	0	0	0	0	1	0	0	1	1	0	0	1	0	0	4
Total Included	0	0	0	1	1	1	2	3	3	5	6	7	4	4	37

After the papers inclusion process, 57 potentially relevant tools were selected for extracting e analyzing the data. Another search was performed on web engines with the particular information of every tool cited by the papers, in order to find more documentation about these tools. There are two goals in this review. The first is concerned with the executable, through which the reviewers could test the functionalities of the tool; and the second involves the written documentation found, i.e. websites, tutorials, technical reports, papers, and dissertations/theses for data extraction.

During date extraction were excluded 18 relevant tools, as a result of applying the detailed exclusion criteria *EC4*, *EC5* and *EC6*. The 39 tools included are listed in Table 3.2 in chronological order. Through this table, it is possible to verify that the number of released tools increased in the years 2005-2009 (with more half of tools released). Table 3.2 summarize the characteristics of each tool: graphical user interface (GUI), prototype (PRT), online (ONL), free (FRE), open-source (OPS), plugin (PLG), source code generator (SCG), tutorials (TUT), example solutions available (ESA), Import/export from/to other applications (I/E), where the tool was developed (DEV). We use "n/a" for information not available.

Additionally, there is an evident lack of tools that support all stages of SPL development and evolution. Based on the systematic review, we found that the majority of the analyzed tools have similar functionalities (project website[5] shows which functionalities each tool supports). In general, the tools available both commercially and freely have the basic functionality for variability managing in SPL, including the manual product configuration functionality. However, we identified that manual method for product configuration may not be sufficient to support industries during SPL managing. The manual process makes product configuration into a complex, time-consuming, and error-prone task. This literature review, along with our expertise, was the rationale for the development of a support method for automatic product configuration. In Sec-

[5]http://homepages.dcc.ufmg.br/~juliana.pereira/SLR/

tion 3.2, we conduct a comparative study between two variability management tools, in order to support extension of the automatic product configuration functionality in one of the tools.

3.1.3 Threats to Validity

External validity concerns the ability to generalize the results to other environments, such as to industry practices [Wohlin et al., 2012]. A major external validity to this study was during the identified primary studies. The search for the tools was conducted in several digital libraries, in order to capture as much as possible the available tools and avoid all sorts of bias. However, the quality of search engines could have influenced the completeness of the identified primary studies. That means our search may have missed those studies whose authors would have used other terms to specify the SPL tool or would not have used the keywords that we used for searches in the title, abstract, and keywords of their papers.

Internal validity concerns the question whether the effect is caused by the independent variables (e.g. reviewers) or by other factors [Wohlin et al., 2012]. In this sense, a limitation of this study concerns the reliability. The reliability has been addressed as far as possible by involving three researchers, and by having a protocol which was piloted and hence evaluated. If the study is replicated by another set of researchers, it is possible that some studies that were removed in this review could be included and other studies could be excluded. However, in general we believe that the internal validity of the literature review is high given the use of a very systematic procedure, consultation with the researchers in the field, involvement, and discussion between three researchers.

Construct validity reflects to what extent the operational measures that are studied really represent what the researcher has in mind [Wohlin et al., 2012]. The three reviewers of this study are researchers in the software engineering field, focused in SPL, and none of the tools was developed by us. Therefore, we are not aware of any bias we may have introduced during the analyses, but it might be possible that the conclusions might have been affected by our personal interest and opinions. From the reviewers perspective, another construct validity threat could be biased judgment. The decision of which studies to include or to exclude and how to categorize the studies could be biased and thus pose a threat. A possible threat in such review is to exclude some relevant tool. To minimize this threat both the processes of inclusion and exclusion were piloted by the three reviewers. Furthermore, potentially relevant studies that were excluded were documented. So we believe that we not have omitted any tool.

Table 3.2. Characteristics each Tool Supports

Tool [Ref.]	Year	GUI	PRT	ONL	FRE	OPS	PLG	SGC	TUT	ESA	I/E	DEV
ISMT4SPL [Park et al., 2012]	2012	•			n/a	n/a		•	•	n/a	•	Academic
LISA toolkit [Groher and Weinreich, 2013]	2012	•			n/a	n/a	•	•	n/a	n/a	•	Academic
Sysiphus [Thurimella and Bruegge, 2012]	2012	•		•	•	•	•		•	n/a	n/a	Academic
Pacogen [Hervieu et al., 2011]	2011				•	•			n/a	•	•	Academic
FMT [Laguna and Hernández, 2010]	2009	•			•	n/a	•		•	•	•	Academic
Hephaestus [Bonifácio et al., 2009]	2009	•			•	•			n/a	•	•	Academic
Hydra [Salazar, 2009]	2009	•			•	n/a	•		•	•	•	Both
SPLOT [Mendonça et al., 2009]	2009	•		•	•	•			•	•	•	Academic
S2T2 [Botterweck et al., 2009]	2009	•	•		•	n/a	•		n/a	•	n/a	Academic
VariaMos [Mazo et al., 2012]	2009	•			•	n/a	•		•	n/a	•	Academic
FeatureMapper [Heidenreich et al., 2008]	2008	•			•	•	•		n/a	•	•	Academic
MoSPL [Thao et al., 2008]	2008	•	•		n/a	n/a			•	n/a	•	Academic
VISIT-FC [Cawley et al., 2008]	2008	•	•		n/a	n/a			n/a	n/a	•	Academic
DecisionKing [Dhungana et al., 2007]	2007	•			n/a	n/a	•		n/a	n/a	•	Both
DOPLER [Dhungana et al., 2011]	2007	•			n/a	n/a	•		n/a	n/a	•	Both
FaMa [Benavides et al., 2007]	2007	•			•	•	•	•	•	•	•	Academic
GenArch [Cirilo et al., 2008]	2007	•			•	•	•		n/a	•	•	Academic
REMAP-tool [Schmid et al., 2006]	2006	•	•		n/a	n/a	•		•	n/a	•	Academic
YaM [Jain and Biesiadecki, 2006]	2006	•		•					n/a	n/a	n/a	Academic
DOORS Extension [Buhne et al., 2005]	2005	•							•	n/a	•	Academic
FeatureIDE [Thüm et al., 2014]	2005	•			•	•	•	•	•	•	•	Academic
Kumbang [Myllärniemi et al., 2007]	2005	•			•	•	•	•	•	•	n/a	Industry
PLUSS toolkit [Eriksson et al., 2005]	2005	•							•	•	n/a	Both
VARMOD [Pohl, 2003]	2005	•	•			n/a	•		n/a	•	•	Academic
XFeature [Ondrej and Alessandro, 2005]	2005	•			•	•	•	•	•	•	•	Both
COVAMOF-VS [Sinnema et al., 2004]	2004	•					•		•	n/a	•	Academic
DREAM [Park et al., 2004]	2004	•							n/a	n/a	•	Academic
ASADAL [Kim et al., 2006]	2003	•			•	n/a		•	n/a	n/a	•	Academic
Pure::Variants [Spinczyk and Beuche, 2004]	2003	•					•	•	•	•	•	Industry
Captain Feature [Bednasch et al., 2003]	2002	•			•	•			•	n/a	n/a	Academic
DECIMAL [Dehlinger et al., 2007]	2002	•			n/a	n/a			•	n/a	n/a	Academic
Odyssey [Braga et al., 1999]	2002	•			•				•	•	n/a	Academic
GEARS [Krueger, 2007]	2001	•					•	•	n/a	n/a	•	Industry
WeCoTim [Asikainen et al., 2004]	2000	•	•	•						•	•	Industry
HOLMES [Succi et al., 2001]	1999	•	•		n/a	n/a			n/a	n/a	•	Academic
DARE [Frakes et al., 1997]	1998	•	•						n/a	n/a	n/a	Industry
Metadoc FM [Thurimella and Janzen, 2011]	1998	•					•		•	•	•	Industry
DOMAIN [Tracz, 1995]	1995	•		•	•	•			•	•	•	Industry
001 [Krut, 1993]	1993	•						•	n/a	n/a	n/a	Academic

Conclusion validity concerns the relation between the treatments and the outcome of the review [Wohlin et al., 2012]. From the reviewers perspective, a potential threat to conclusion validity is the reliability of the data extraction categories from the tools, since not all the information was obvious to answer the research question and some data had to be interpreted. Therefore, in order to ensure the validity, multiple sources of data were analyzed, i.e. papers, prototypes, technical reports, manuals, and tools execution. Furthermore, in the event of a disagreement between the two primary reviewers, the third reviewer acted as an arbitrator to ensure agreement was reached.

3.2 Comparative Study

Based on the systematic literature review of tools to support SPL variability management, our review suggests that there are opportunities of development of a method to support automatic product configuration. This section presents the results of a comparative study detailed of two feature modeling tools. The goal of this study is to investigate the options of feature modeling tools to be extended with the automatic product configuration functionality. This study involved 56 developers taking an advanced Software Engineering course. In this study, we performed a four-dimension qualitative analysis targeting at: (i) feature model editor, (ii) automated analysis of feature models, (iii) product configuration, and (iv) notation used by the tools for features modeling. In this dissertation, we present the more relevant results (feature model editor and product configuration) and further analyses and details about this comparative study can be found in our paper [Pereira et al., 2013]. The next sections are organized as follows. Section 3.2.1 presents the justification for choosing the tools; the summary of the background information of participants that took part in this study; and an explanation of the training session and tasks assigned to each participant. Section 3.2.2 reports and analyzes the results of this comparative study and motivates the extension of one of the analyzed tools. Finally, Section 3.2.3 presents the threats to validity related for this comparative study and how they were addressed prior of the study to minimize their impact.

3.2.1 Study Settings

SPLOT [Mendonça et al., 2009] and FeatureIDE [Thüm et al., 2014] were selected for this study based on the systematic literature review presented in Section 3.1. Based on this study, we discard all tools that are only prototypes (Table 3.2). We also aimed to select plugin and open source tools, because plugin provides the extension

of tools already established and known, and open source tools allow its extension. On the other hand, proprietary tools could hinder some sorts of analyses, well as its extension. For instance, we do not have access to all the functionalities of the tool. In a second analysis, we also include only tools with tutorials and example solutions available, because the tutorial and example solutions are important for users that do not have previous training before the tool usage. Moreover, we include tools with functionality import/export from/to other applications. This characteristic concerns the interoperability between other applications, allowing users to migration and the use of additional functionality not provided by the tool. After applying these filters, four tools were selected (SPLOT, FaMa, FeatureIDE, and XFeature). Finally, we verify the key functionalities provided by typical feature modeling tools, such as to create and edit a feature model (*domain representation*, *variability*, *mandatory features*, and *composition rule*), automatically analyze the feature model (*reports* and *consistency check*), and product configuration (*manual product configuration*). After this filter, we selected three tools (SPLOT, FaMa, FeatureIDE). Next, we decide to select FeatureIDE and SPLOT, because they are mature tools that we have experienced and that we have contact with the developers of these tools. Table 3.3 summarizes the main functionality of both SPLOT and FeatureIDE tools.

Table 3.3. Main Functionalities SPLOT and FeatureIDE

Functionality	**SPLOT**	**FeatureIDE**
Feature model editor	✓	✓
Automated feature model analysis	✓	✓
Manual product configuration	✓	✓
Tutorials and example solutions	✓	✓
Available online	✓	
Integration with code		✓
Feature model notation	tree	tree and diagram

SPLOT [Mendonça et al., 2009] is a Web-based, free and open source project. At the tool website , we can find a repository of more than 400 feature models created by tool users over four years. Additionally, it also provides a standalone tool version that can be installed in a private machine. On the other hand, FeatureIDE [Thüm et al., 2014] is a standalone project implemented as an Eclipse plugin. FeatureIDE is integrated with several programming and composition languages with a focus on development for reuse. It was developed to support both aspect-oriented [Kiczales et al., 2001] and feature oriented programming [Batory, 2005]. While FeatureIDE widely

covers the SPL development process, SPLOT does not provide means for generation or integration of code.

Participants involved in this study are 56 young developers taking an advanced software engineering course. All participants are graduated or close to graduate since the course targets post-graduated MSc and PhD students. To avoid biasing the study results, each participant only took part in one study semester and only used one tool, either SPLOT or FeatureIDE. Table 3.4 shows that FeatureIDE was used by 27 participants being 6 in the first and 21 in the second semester. Additionally, SPLOT was used by 29 participants being 15 in the first and 14 in the second semester. Each participant worked individually to accomplish the study tasks. Participants in the 2011 first semester are named *T1-S01* to *T1-S06*; in the 2011 second semester, *T2-S01* to *T2-S21*; in the 2012 first semester, *T3-S01* to *T3-S15*; and in the 2012 second semester, *T4-S01* to *T4-S14*.

Table 3.4. Background of Participants

Tool	**FeatureIDE**		**SPLOT**		**No Answer**
Semesters	**T1 (2011-1)**	**T2 (2011-2)**	**T3 (2012-1)**	**T4 (2012-2)**	
Work Experience	S01, S03, S04, S06	S02, S04, S05, S07, S14, S18	S04, S07, S09, S10, S14, S15	S01-S03, S05-S12	T1: S02
UML Design	S01, S03, S04 - S06	S02, S04, S05, S08, S14, S18, S20	S03, S04, S07-S10, S12, S14, S15	S02, S04, S05, S08-S12	T2: S13, S16
Java Programming	S01, S03, S04 - S06	S01, S02, S04-S06, S08-S12, S14, S15, S17, S18, S20, S21	S03, S04, S07-S10, S12, S14, S15	S01-S03, S05-S12	T3: S01, S02, S05, S06, S11, S13
# of Participants	6	21	15	14	9

Before starting the experiment, we used a background questionnaire to acquire previous knowledge about the participants. Table 3.4 summarizes knowledge that participants claimed to have in the background questionnaire with respect to work experience, UML design, and Java programming. Second, third, fourth and fifth columns in this table show the participants who claimed to have knowledge medium or high in a particular skill. Answering the questionnaire is not compulsory and participants who have not answered it are annotated in the last column (No Answer). However, although some cases participants who chose not to answer the questionnaire, we observe in Table 3.4 that, in general, all participants have at least basic knowledge in software development and technology.

We conducted a 1.5 hour training session, where we introduced participants not only to the analyzed tools but also to the basic concepts of feature modeling and SPL. The same training session (with the same content and instructor) was performed

in all four groups (2011-1, 2011-2, 2012-1 and 2012-2). After the training session, we asked participants to perform some tasks using either SPLOT or FeatureIDE (see Table 3.4). The tasks include: (i) feature model creation, (ii) feature model edition, (iii) automated feature model analysis, (iv) product configuration and (iv) feature model save. The whole study was performed in a computer laboratory with 25 equally configured equipments. We ask participants to answer a questionnaire with two simple questions about functionalities of the tool that they like and dislike. The tasks and questionnaire aimed to verify the basic functionality provided by the tools, in order to evaluate the best tool in terms of usability for extension. The questionnaire and all answers are available in the project website[6].

3.2.2 Results and Analysis

Both FeatureIDE and SPLOT present different types of feature models as show in Figure 3.2. With respect to feature model editor in FeatureIDE, even less experienced participants think the editor interface of FeatureIDE is simple and easy to use. For instance, *T2-S17* said that "*even without seeing the tutorial, I used it and performed all tasks right from the first time*". Similar comments were made by other participants, such as *T2-S21* who stated that "*the graphical representation of the feature model is organized and facilitates visualizing the configuration space*". Two positive functionalities were cited: namely automatic organization of features and shortcuts for the mostly used functions. For instance, with respect to the automatic organization of features, participant *T1-S01* stated that "*the tool allows a nice view of the feature model because when we insert new features it automatically adjusts spaces between boxes to keep everything on screen*". participant *T1-S01* also observed that "*shortcuts make it easy to switch between mandatory, optional, and alternative features*". Following the same trend, participant *T2-S6* concludes that "*shortcuts helped to speed up the feature model creation*".

With respect to SPLOT, participants complained about the lack of support to restructure a feature model. For instance, participant *T3-S12* said that "*there is no way to manually reorder features of a tree by dragging and dropping them*". A similar observation is made by participant *T3-S02*: "*one cannot create an OR or XOR group based on preexisting features*". The Web interface of SPLOT also led some usability-related issues. One of these issues was raised by participant *T3-S3* saying that "*the enter button does not work to save changes in a feature; you need to click outside a feature box to validate your changes*". Besides the *enter* button, another participant

[6]http://homepages.dcc.ufmg.br/~juliana.pereira/spl_study/

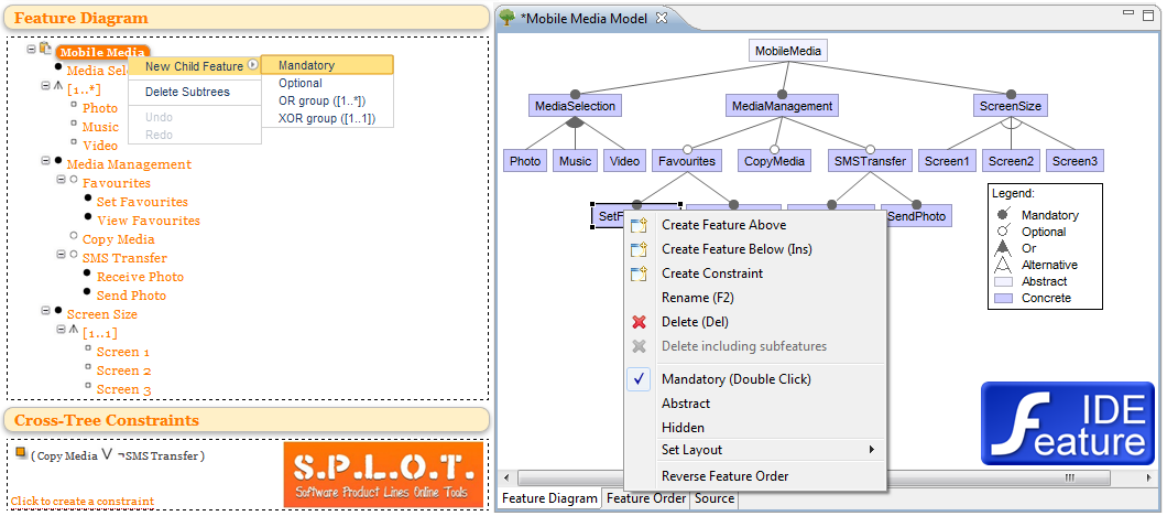

Figure 3.2. Feature Model Editor SPLOT and FeatureIDE

also spots an issue with the *delete* button. Participant *T3-S3* stated that "*a feature is not removed by pressing the delete button*". In other words, participants of this study observed that, in general, shortcuts work fine in FeatureIDE, but it is a weak aspect in SPLOT.

It is notable that features models are continually increasing in size and complexity. Therefore, it is required support for product configuration. Product configuration was performed in both tools by means of the manual functionality as shown in Figure 3.3. FeatureIDE allows to create multiple configurations and to set the default one. On the other hand, SPLOT allows a single product configuration to be created and it does not allow saving it in the repository. Additionally, SPLOT also requires the user to save the feature model on its repository and use an URL for product configuration. Participant *T3-S13* points out this fact as a drawback by saying "*if the user does not generate the URL for the feature model, he cannot use the product configuration option*".

In general, SPLOT and FeatureIDE can be fairly used to represent feature models since they have the main model elements required to model the domain. The SPLOT users have been widely praised in relation to the automated analysis and display of statistics about feature models. On the other hand, several participants indicated usability-specific issues in this tool, such as inability to rearrange the features inside an existing model, difficulty in finding buttons to save the model, and lack of privacy. Models created by SPLOT are automatically stored into the tool repository available to all tool users. In fact, SPLOT and FeatureIDE are two very different tools, although both tools provide support for feature modeling. SPLOT is a tool to be readily acces-

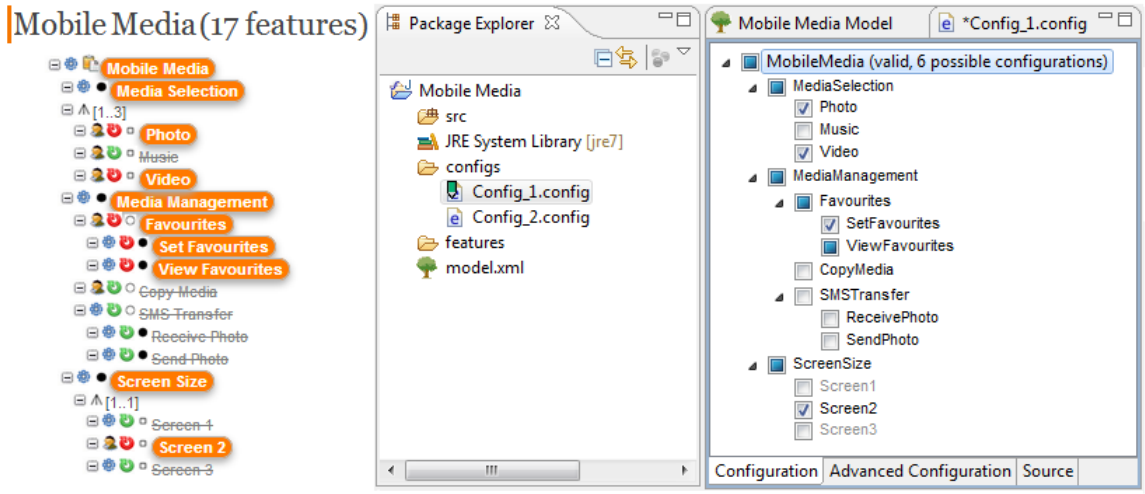

Figure 3.3. Product Configuration SPLOT and FeatureIDE

sible to those interested only on SPL modeling and analysis (before of the development process) and FeatureIDE is a tool focused on integration with the development process. It presents integration with others tools, code generation, and supports different languages for SPL implementation, such as, AspectJ [Kiczales et al., 2001] and AHEAD [Batory, 2005]. Moreover, it is notable that FeatureIDE provides a greater support for product configuration functionalitty. Therefore, taking into consideration our need and purpose of use, we choose extend the FeatureIDE tool. In addition to the strengths of the tool be greater, it allows automatic code generation. Subsequently, FeatureIDE covers the whole development process SPL.

3.2.3 Threats to Validity

External validity concerns the ability to generalize the results to other environments, such as to industry practices [Wohlin et al., 2012]. A major external validity to the comparative study can be the selected tools and participants. We choose two tools, among many available, and we cannot guarantee that our observations can be generalized to other tools. Moreover, previous experiences with SPL or with some of the tools used were not taken into consideration. Therefore, we cannot generalize the results because these experiences could positively influence the results obtained.

Internal validity concerns the question whether the effect is caused by the independent variables (e.g. level of knowledge) or by other factors [Wohlin et al., 2012]. In this sense, a limitation of this study concerns the absence of balancing the participants in groups according to their knowledge. It can be argued that the level of knowledge of some participant may not reflect the state of practice. To minimize this threat, we pro-

vide 1.5 hour training session to introduce participants to the basic required knowledge and a questionnaire for helping the better characterize the sample as a whole. Additionally, 1.5 hour training session may not have been enough for subjects that begin without much knowledge and being exposed for the first time to these tools. However, Basili et al. [1999] and Kitchenham et al. [2002] argue that even less experienced participants can help researchers to obtain preliminary, but still important evidence that should be further investigated in later controlled experiments.

The construct and conclusion threats may have occurred in the formulation of the questionnaire or during the interpretation of the results by the researchers since our study is mainly qualitative. Due to this qualitative nature, data are not suitable for quantitative analysis and statistical tests. As far as we are concerned, this is the first experiment conducted to analyze and compare these tools. To minimize this threat, we cross-discuss all the experimental procedures. Basili et al. [1999] and Kitchenham et al. [2002] argue that qualitative studies play an important role in experimentation in software engineering.

3.3 Tool Support

Based on the results of our experiments, we decided to implement a tool, called SPLConfig, to support automatic product configuration. The main goal of the SPLConfig tool is to derive an optimized feature selection that satisfies an arbitrary set of requirements. The primary contribution of this tool is to assist industries during product configuration, answering the following question: What is the set of features that balances cost and customer satisfaction, based on available budget? By resolving this problem, industries can more effectively achieve greater customer satisfaction. Section 3.3.1 presents the SPLConfig architecture and Section 3.3.2 discusses its design and implementation.

3.3.1 SPLConfig Architecture

SPLConfig is an Eclipse plugin implemented in Java. It requires one additional plugin, named FeatureIDE, in order to support product configuration. Figure 3.4 presents the architecture of SPLConfig and its relationships with FeatureIDE and Eclipse platform. The decision of extending FeatureIDE is particularly attractive for two reasons. First, FeatureIDE is open source, extensible, modular, and easy to understand its code. Second, our motivation in extending FeatureIDE occurred because we could reuse the key functionality of typical feature modeling tools, such as to create and edit a fea-

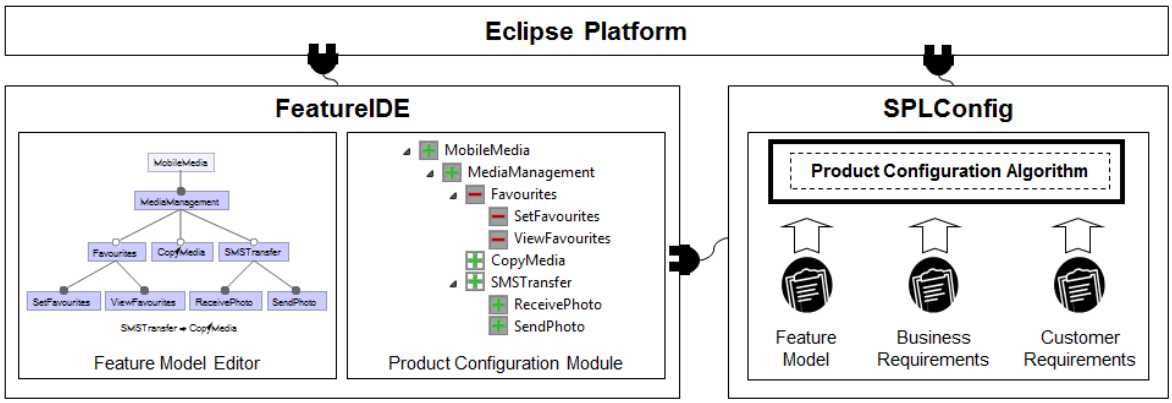

Figure 3.4. SPLConfig's Architecture

ture model, to automatically analyze the feature model, product configuration basic infrastructure, code generation, and import/export feature model. FeatureIDE is a development tool for SPL largely used and it supports all phases of feature-oriented software development of SPLs: domain analysis, domain implementation, requirements analysis, and code generation.

The implementation of an automatic product configuration allows the customization of different products according to the business and customer requirements, and composition constraints of the feature model (feature-tree and cross-tree constraints). In this tool extension, we have considered benefit of the features and total budget as customer requirements, and cost of each feature as business requirements. In Figure 3.4 two important activities are performed in the SPL lifecycle, domain engineering and application engineering, described below:

Domain Engineering. This process is represented by a feature model and is supported by FeatureIDE (feature model editor in Figure 3.4). Common and variable requirements of the product line are elicited and documented. The developers create reusable artifacts of a product line in such a way that these artifacts are sufficiently adaptable (variable) to efficiently derive individual applications from them.

Application Engineering. From meetings, the developers identify and describe the requirements prioritization of customer (SPLConfig in Figure 3.4). SPLConfig prioritize and create reports to aid the product builder when defining a product configuration for a specific customer. Our product configuration algorithm uses an optimization scheme and provides a valuable decision support to product configuration within the SPL. The product configuration result is visualized in FeatureIDE (product configuration module in Figure 3.4). It is important to observe that the tool can be easily

extended - by means of new algorithms - to support additional non-functional features. Chapter 4 presents algorithms implemented to optimization of the product configuration.

As show in Figure 3.4 an early stage, feature modeling enables SPL scoping, i.e., deciding which features should be supported by an SPL and which should not. This result is a feature model represented in FeatureIDE. In a second stage, during the product configuration, it is necessary to choose the features that appropriately fulfill the business and customer requirements. The variables features are defined as the set of features that can implement the customer requirement. This stage allows a single product configuration to be created through search-based algorithms in SPLConfig. This product configuration is presented by FeatureIDE in the product configuration module. In a third stage, SPLConfig allows manual configuration of features if necessary. In a fourth stage, it also allows compiling and building the product.

3.3.2 Design and Implementation Decisions

This section discusses some design and implementation decisions we made in the development of SPLConfig. Figure 3.5 shows a screenshot of SPLConfig view in the Eclipse IDE. Figure 3.5a) show the package explorer view typical of the Eclipse IDE. Figure 3.5b) and c) illustrates the feature model editor of the FeatureIDE in diagram and outline views, respectively. Figure 3.5d) show SPLConfig view detailed in Figure 3.6. It gives the product builder, a method to automate the product configuration.

Figure 3.6 presents the main view of SPLConfig integrated with the Eclipse IDE. At the top of the view, we have two fields named *Budget* and *Customer* that should be filled by the customer with the available budget and with the customer identification respectively. Each feature in the feature model is presented in a row of this view. Columns give additional feature information, such as its name, the level of importance ($Benefit$), and the cost of development of each feature ($Cost$). Moreover, this view includes typical buttons, such as $Refresh$ and $Execute$, which trigger their respective actions. $Refresh$ is used to update data being presented in this view, while $Execute$ selects the most appropriate product configuration that best satisfies the customer requirements. Each product configuration generated is presented by FeatureIDE view.

In addition to this view, we extend Eclipse with a preference page (Figure 3.7). Industries can set specific preferences, as the cost of development of each feature that makes up the SPL. Therefore, for the same SPL, several products can be generated according to needs and constraints specific of each customer, but keeping the cost of each fixed feature. Note that the FeatureIDE tool view (product configuration module)

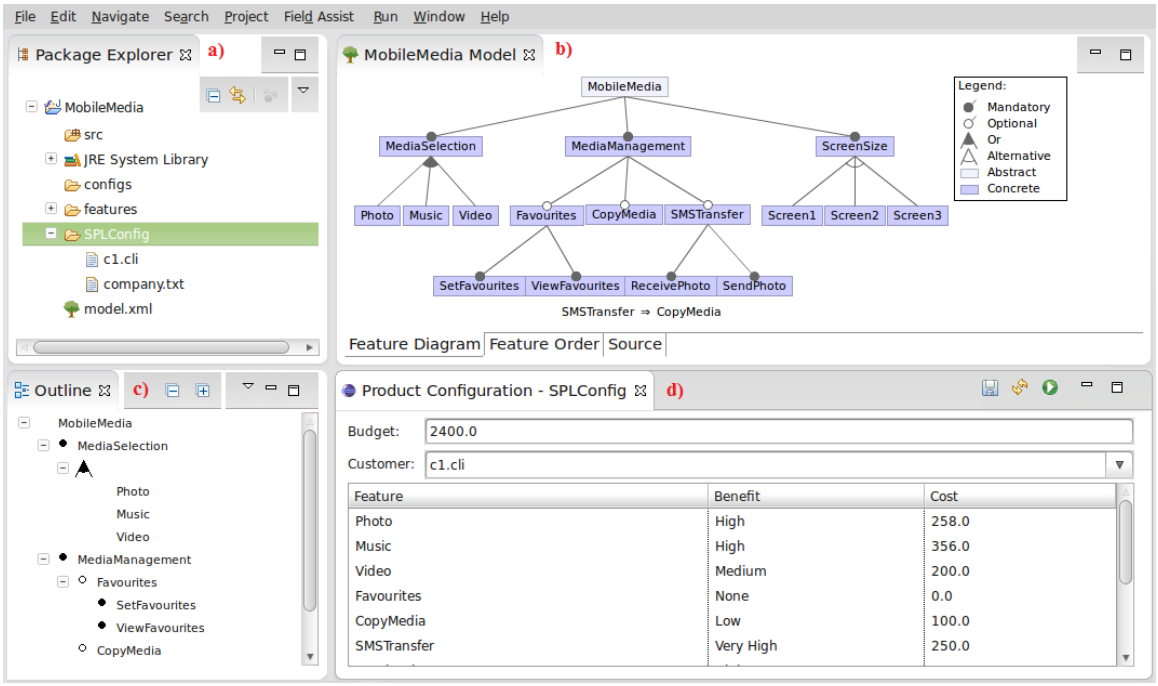

Figure 3.5. SPLConfig View in the Eclipse IDE

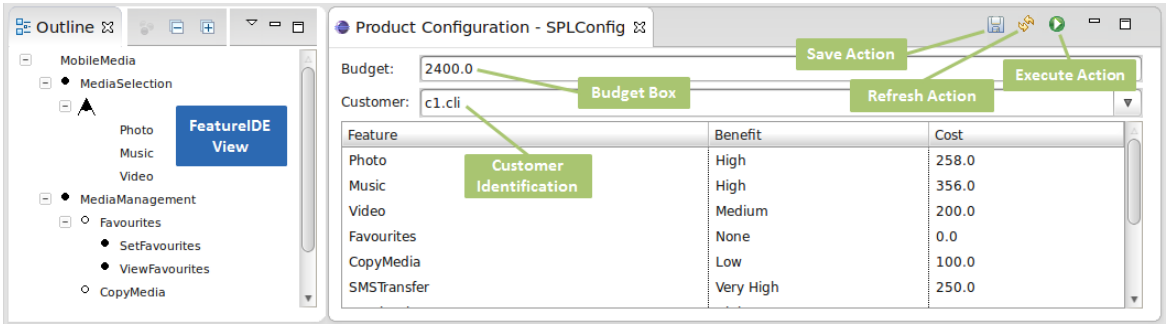

Figure 3.6. SPLConfig Main View

shows the product configuration that best satisfies the customer requirements, but nothing prevents other features from being included or excluded in the final product. Subsequently, the code of the product can be generated and compiled by FeatureIDE.

3.4 Concluding Remarks

In this chapter, we provided a systematic literature review of SPL management tools. The results reveal the chronological backgrounds of various SPL tools. Our proposal contributes specifically with relevant information to checking attributes and requirements relevant to those interested aimed at developing new tools. In addition, we

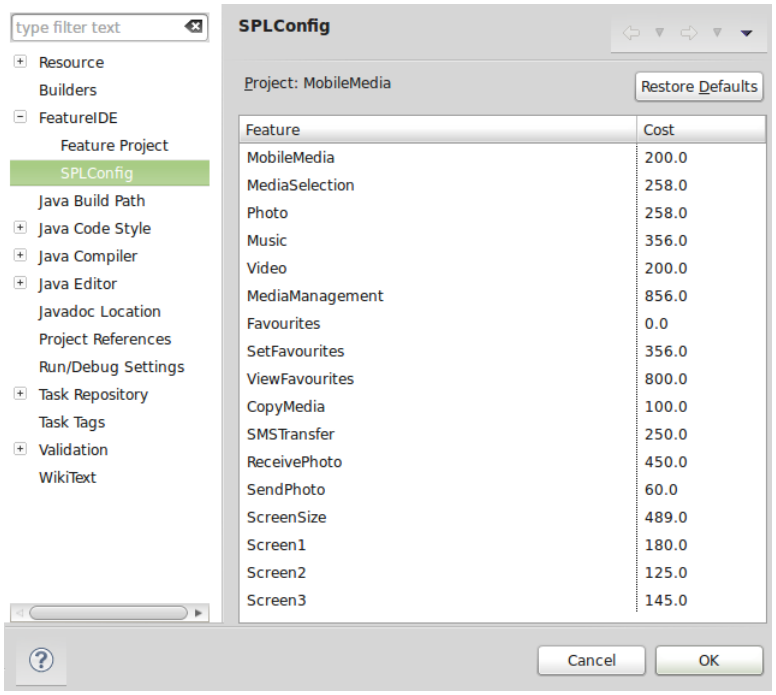

Figure 3.7. SPLConfig Preference Page

conduct a qualitative analyzes with 56 young developers, in which we investigate the strengths and weaknesses of two tools. We discussed the unfeasibility of the manual product configuration process and motivation to choose one of the tools for extension with automatic product configuration. Finally, we present SPLConfig, a developed tool for automatic product configuration in SPL by the use of search-based algorithms. We summarized the method behind of the tool and its main functionalities. In the next chapter, we present the search-based algorithms that are integrated in the developed tool in order to optimize product configuration.

Chapter 4

Search-Based Algorithms for Product Configuration

This chapter presents search-based algorithms for optimizing product configuration in SPLs. In Section 4.1, SPL product configuration is modeled as an optimization problem. This problem aims at maximizing the customer satisfaction, subject to business and customer requirements. It is NP-hard as it contains a SAT [Cormen et al., 2009] as well as a Knapsack [Gallo and Simeone, 1989] sub-problem, both well-known NP-hard problems. A preprocessing algorithm to reduce the size of the feature model is proposed in Section 4.2. Then, an exact backtracking algorithm is proposed in Section 4.3. As the worst case complexity of the latter grows exponentially with the number of features, a greedy heuristic algorithm is proposed in Section 4.4.

4.1 Problem Definition

SPL systems need to generate and evaluate different product configurations that satisfy a set of business and customer requirements, in order to assist the user to find the product configuration that best attends their needs to. Explicitly generate all valid product configurations and evaluate all of them may be impractical as the number of configurations grows exponentially with the number of features. The problem of SPL product configuration (SPL-PC) proposed here is described as follows. The decision if a feature is selected or not is modeled by variable $x_i \in \{0, 1\}$ such that

$$x_i = \begin{cases} 1 & \text{if feature } i \text{ is selected} \\ 0 & \text{otherwise} \end{cases}$$

The constants defined by the model are the following:

- $T = (V, A)$ is a rooted tree where V is a set of nodes, which represent features, and A is a set of arcs such that $(i, j) \in A$ denotes that j can only be selected if i is also selected. T is referred as the feature tree and j is said a sub-feature of i.

- $M \subseteq V$ is the set of mandatory features, such that the root node $r \in M$.

- $O \subset V$ is the set of optional features, such that $O \cup M = V$ and $O \cap M = \emptyset$.

- $XOR_i \subset 2^V$ is the set of all exclusive alternatives rooted in $i \in V$, such that if $Q \in XOR_i$ then $(i, s) \in A$, for all $s \in Q$. Moreover, if $R \in (XOR_i$ or $OR_i)$ then $s \notin R$. $XOR_i = \emptyset$ if there is no exclusive alternative rooted on i.

- $OR_i \subset 2^V$ is the set of all non-exclusive alternatives rooted in $i \in V$, such that if $Q \in OR_i$ then $(i, s) \in A$, for all $s \in Q$. Moreover, if $R \in (XOR_i$ or $OR_i)$ then $s \notin R$. $OR_i = \emptyset$ if there is no non-exclusive alternative rooted on i.

- $c_i \in \mathbb{R}$ is the cost of the feature $i \in V$.

- $D \in \mathbb{R}$ is the available customer budget.

- $b_i \in \mathbb{N}$ is the level of importance of the feature $i \in V$, with $b_i = 0$ for all $i \in M$, and $b_j \leq b_i$, for all $(i, j) \in A$.

- $E = (E_1(x), E_2(x), ...)$ is the set of cross-tree constraints, where $E_j(x)$ is a logical expression over the variables x. It admits conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), implication ($\rightarrow$), and biconditional ($\leftrightarrow$) operations. For example, $E_1(x) = (x_1 \wedge x_2) \vee x_3 \rightarrow x_4$ implies that if *Feature 1* and *Feature 2* are simultaneously selected or if *Feature 3* is selected, them *Feature 4* must also be select.

Given the variables and constants described above, SPL-PC is defined as follows:

Maximize

$$F(x) = \sum_{i \in V} b_i x_i \tag{1}$$

Subject to:

$$\sum_{i \in V} c_i x_i \leq D \tag{2}$$

$$x_p = 1 \rightarrow x_i = 1 \qquad (p,i) \in A : i \in M \tag{3}$$

$$x_p = 0 \rightarrow x_i = 0 \qquad (p,i) \in A : i \in V \setminus \{r\} \tag{4}$$

$$x_p = 1 \rightarrow \sum_{i \in Q} x_i = 1 \qquad \forall p \in V : XOR_p \neq \emptyset, \forall Q \in XOR_p \tag{5}$$

$$x_p = 1 \rightarrow \sum_{i \in Q} x_i \geq 1 \qquad \forall p \in V : OR_p \neq \emptyset, \forall Q \in OR_p \tag{6}$$

$$E_j(x) = true \qquad \forall E_j(x) \in E \tag{7}$$

SPL-PC consists in to maximize the customer satisfaction (1) subject to the budget constraints (2), the feature-tree constraints (3) to (6), and the cross-tree constraints (7). Constraint (3) ensures that the mandatory feature $i \in M$ is selected if its parent $p \in V$ is selected. Constraint (4) ensures that the feature $i \in V \setminus \{r\}$ is not selected if its parent $p \in V$ is not selected, where r is the root of T. Constraint (5) ensures that, for each exclusive alternative set Q rooted on $p \in V$, exactly one feature in Q is selected if p is selected. Constraint (6) ensures that, for each non-exclusive alternative set Q rooted on $p \in V$, at least one feature in Q is selected if p is selected. Finally, constraint (7) ensures that, all cross-tree constraints are satisfied. Previous research has shown that finding valid product configurations that conforms to the composite constraints and the cross-tree constraints, i.e. (3) to (7), is NP-hard [Mendonça et al., 2009]. Besides, (1) and (2) consists of the classic Knapsack problem [Gallo and Simeone, 1989], which is NP-hard. In addition, (1) and (7) can be reduced to the maximum satisfiability problem (MAX-SAT) [Cormen et al., 2009] which is also NP-hard. Therefore, SPL-PC is also hard to approximate, since even to find a feasible solution is NP-Complete because a satisfiability problem must be solved in order to find any feasible solution.

4.2 Preprocessing Algorithm

In order to reduce the size of the feature model, the preprocessing procedure merges and removes nodes in the feature tree, reducing the number of features in T. The algorithm is divided in three parts. First, the mandatory features are merged with their parents. Second, it attempts to fix the value of variables x_i, for all $i \in O$, to 0 or to 1, by checking the cross-tree constraints (7) and the budget constraint (2). Third, for each exclusive and non-exclusive alternative set, it checks if after the previous parts

all but one feature remains in this set. If so, this single feature is merged with its parent.

The pseudo-code of the preprocessing algorithm is shown in Algorithm 4.1. It takes as input the feature tree $T = (V, A)$, the sets M, O, E, OR_v, and XOR_v, for all $v \in V$, as well as the budget D, the cost c_v, and level of importance b_v for each feature $v \in V$. The algorithm returns *true* if it successfully finishes or $false$ if the instance turns out infeasible.

Algorithm 4.1 $Preprocess(T = (V, A), M, O, E, OR, XOR, D, c, b) \mapsto \{true, false\}$

```
1: let r ← root(T)
2: for each v ∈ M \ {r} do
3:     let p ← parent(v)
4:     merge v into p and update T, M, OR, XOR, c_p, E
5: end for
6: if č(r) > D then
7:     return false
8: end if
9: for each v ∈ O do
10:    let p ← parent(v)
11:    if Impossible(E, v, 1) and Impossible(E, v, 0) then
12:        return false
13:    else if Impossible(E, v, 1) or č(v) + ĉ(p) > D then
14:        remove v from T and update T, O, OR, XOR, E
15:    else if Impossible(E, v, 0) then
16:        merge v into p and update T, O, OR, XOR, c_p, b_p, E
17:    end if
18: end for
19: for each Q ∈ OR_p ∪ XOR_p with p ∈ V do
20:    if |Q| = 1 then
21:        let q be the only node in Q
22:        merge q into p and update T, O, OR, XOR, c_p, b_p, E
23:    end if
24: end for
25: return true
```

In the first part, the loop of lines 2 to 5 is repeated for each mandatory feature, except for the root of T. Let the feature $r \in V$ be the root of T (line 1). For each mandatory feature $v \in M \backslash \{r\}$ (line 2), v is merged with its parent $p \in V$ and T, M, OR, XOR, c_p, and E are updated in line 4 as follows: (i) v and p are contracted in T, (ii) $M = M \backslash \{v\}$, (iii) $OR_p = OR_p \cup OR_v$, (iv) $XOR_p = XOR_p \cup XOR_v$, (v) $c_p = c_p + c_v$, and (vi) variable x_v is replaced by variable x_p in all expressions in E, if $p \neq r$, or by 1 otherwise. The value of b_p does not need to be updated, because $b_v = 0$

as v is a mandatory feature. After this update, the node p represents two features. Then, variable $x_p = 1$ denotes that both features are selected, and $x_p = 0$ denotes that both features are not selected. For sake of simplicity, in the remainder of the this text we do not detail the data structure that keeps track of which features are merged features or single features, and we assume that we have available a function $features(v)$ that returns a set with all the features from the original feature tree that were merged in $v \in V$. At the end of this part, all the features in V are optional, except the root feature.

In the second part (lines 6 to 18), the budget constraints (2) and cross-tree constraints (7) are checked for each feature $v \in V$, in order to (i) prove the instance is infeasible, (ii) remove features that cannot be selected without violating (2) and (7), or (iii) merge (with the respective parent) features that cannot be left unselected, if their parents are selected, without violating (2) and (7). We denote by $\hat{c}(v) = c_v + \hat{c}(p)$ a recursive function that returns the sum of the cost of all features from $v \in O$ to the root r of T, where p is the parent of v and $\hat{c}(r) = c_r$. Besides, we denote by $\check{c}(v) = c_v + \sum_{X \in OR_v \cup XOR_v} (min_{u \in X} \check{c}(u))$ a recursive function that returns the minimum cost for selecting a feature $v \in V$ without violating the composite constraints (5) and (6). In line 6, the procedure checks if $\check{c}(r)$ fits the budget. If it does not, no feasible solution exists and the algorithm returns $false$ in line 7. Following, the loop of lines 9 to 18 is repeated for each optional feature $v \in O$. Let $p \in V$ be the parent of v, and $Impossible(E, v, \beta)$ be a function that returns $true$ if it is impossible to satisfy any expression $E_j(x) \in E$ by replacing the value of x_v by $\beta \in \{0, 1\}$, and $false$ otherwise. We note that this function does not solve a satisfiability problem, as it only checks the expressions in E in which the values of all variables are already fixed, except that of x_v. Three cases are checked in the loop of lines 9 to 18. In the first case (lines 11 and 12), if constraints (7) cannot be satisfied by fixing x_v neither to 0 or to 1, then the procedure returns $false$ in line 12, indicating that the original instance is infeasible. In the second case (lines 13 and 14), if making $x_v = 1$ violates the cross-tree constraints (7) or the budget constraint (2), then v is removed from the feature model and T, O, OR, XOR, and E are updated in line 14 as follows: (i) the subtree rooted on v, denoted by $T[v] = (V[v], A[v])$, is removed from T, (ii) $O = O \backslash V[v]$, (iii) $Q = Q \backslash \{q\}$ for all $u \in V$, $Q \in OR_u \cup XOR_u$, $q \in V[v]$, and (iv) for all $q \in V[v]$, variable x_q is replaced by 0 in every expression in E. In the third case (lines 15 and 16), if making $x_v = 0$ violates the cross-tree constraints (7), then v is merged with its parent $p \in V$ and T, O, OR, XOR, c_p, b_p, and E are updated in line 16 as follows: (i) v and p are contracted in T, (ii) $O = O \backslash \{v\}$, (iii) $OR_p = OR_p \cup OR_v$, (iv) $XOR_p = XOR_p \cup XOR_v$, (v) $c_p = c_p + c_v$, (vi) $b_p = b_p + b_v$, and (vii) variable x_v is replace by variable x_p in all

expressions in E, if $p \neq r$, or by 1 otherwise. Besides, (viii) if there is a non-exclusive alternative set $Q \in OR_p$ such that $v \in Q$, then $OR_p = OR_p \backslash Q$ because this set is satisfied by v, and (ix) if there is an exclusive alternative set $Q \in XOR_p$ such that $v \in Q$, then $XOR_p = XOR_p \backslash Q$ and each node $q \in Q$ is removed from the feature model the same way as in line 14. At the end of this part all the optional features can be selected individually without violating any constraint.

In the third part, the loop of lines 19 to 24 is repeated for all $Q \in OR_p \cup XOR_p$ with $p \in V$. If $|Q| = 1$ (line 20), then the single feature $q \in Q$ is merged with its parent p and T, O, OR, XOR, c_v, b_v, and E are updated in line 22 the same way as they were in line 16. Finally, in line 25, the algorithm returns *true* meaning that the instance was successfully preprocessed.

4.3 Backtracking Algorithm

Backtracking is a refined search algorithm for implicitly enumeration, where infeasible and sub optimal solutions are eliminated without being explicitly examined [Cormen et al., 2009]. It can compute the number of valid configurations considerably faster than an exhaustive enumeration of all the $2^{|O|}$ possible features combinations [Mendonça et al., 2009]. The fundamental principles of backtracking algorithms are (i) the order of evaluating the variables and (ii) the order of assigning the values to each variable. In the backtracking proposed in this section, the features in V are sorted by the order they are visited by a breath first search in T started from its root feature. This way, whenever a variable is evaluated, the values of the variable correspondent to its parent are already fixed. Variables are first assigned to the value 1, and then to the value 0.

The pseudo-code of the backtracking procedure is shown in Algorithm 4.2. Let $X \in \{0,1\}^{|V|}$ be an array representing a solution for SPL-PC, where $X[v]$ keeps the value of variable x_v. Let also $F(X)$ be the evaluation of X in the objective function (1), with $F(X) < 0$ if X is infeasible. The algorithm takes as input the feature tree $T = (V, A)$, the sets M, O, E, OR_v, and XOR_v, for all $v \in V$, as well as the budget D, the cost c_v and the level of importance b_v of each feature $v \in V$. Moreover, it receives the permutation π that defines the order in which the variables correspondent to the features in V are assigned values, the index i that defines the next feature in π to be evaluated, the current (possibly infeasible) solution X, and the best known solution X^*. The backtracking procedure is first executed with $i = 2$, $X[1] = 1$, and $X^*[1] = 1$, because π_1 is always the root feature. We note that this procedure can be executed after the preprocessing procedure of Algorithm 4.1 or not.

Algorithm 4.2 $Backtracking(T = (V, A), M, O, E, OR, XOR, D, c, b, \pi, i, X, X^*) \mapsto X^*$

```
1: if i = |V| then
2:     if X is feasible and F(X) > F(X*) then
3:         return X
4:     else
5:         return X*
6:     end if
7: else
8:     let p ← parent(π_i)
9:     if c_{π_i} ≤ D and X[p] = 1 and ∄Q ∈ XOR_p : v ∈ Q ∧ X[v] = 1 and Possible(E, π_i, 1)
   then
10:        X[π_i] ← 1
11:        X* ← Backtracking(T, M, O, E, OR, XOR, D − c_{π_i}, c, b, π, i + 1, X, X*)
12:    end if
13:    if (π_i ∈ O or X[p] = 0) and Possible(E, π_i, 0) then
14:        X[π_i] ← 0
15:        X* ← Backtracking(T, M, O, E, OR, XOR, D, c, b, π, i + 1, X, X*)
16:    end if
17: end if
```

In the base case (lines 1 to 6), all variables in X were assigned values. In this case, if the current solution X is feasible and better than X^* (line 2), the procedure returns X in line 3, otherwise it returns X^* in line 5.

The recursive case consists of lines 8 to 16. The parent p of the next feature π_i to be evaluated is identified in line 8. In line 9, the algorithm checks if $\pi_i \in V$ can be selected. That is: (i) if the cost of π_i fits the budget, (ii) if its parent $p \in V$ has been selected, (iii) if there is no set $Q \in XOR_p$ such that $\pi_i \in Q$ and there is another feature $v \in Q$ already selected in this exclusive alternative set, and (iv) if constraints (7) can be satisfied by fixing π_i to 1. If so, feature π_i is selected in line 10, and the backtracking procedure is recursively called white $i + 1$ in line 11. In line 12, the algorithm checks if $\pi_i \in O$ can be left unselected. That is: (i) if π_i is an optional feature or, (ii) if its parent was not selected, and (iii) if constraints (7) can be satisfied by fixing π_i to 0. If so, feature π_i is not selected in line 14, and the backtracking procedure is recursively called in line 15.

4.4 Greedy Heuristic Algorithm

The objective of the greedy heuristic is to find a feasible solution that maximizes the objective function (1). Any subset $S \subset V$ of features that satisfies the constraints (2) to (7) is called a feasible solution. The heuristic proposed in this section for SPL-PC works in steps. In each step, the optimal local decision is made and one feature

is selected, expecting that this choice leads to a global optimum (or near optimal) solution. Before each step, the feature tree is preprocessed, and one sub-feature $v \in O$ of the root feature r is selected and merged with the root. We note that, after the preprocessing: $V = O \cup \{r\}$ and all features remaining in O can be individualy selected without violating any constraint. The procedure stops when $O = \emptyset$. At this point, all features selected have been merged with the root. We point out that there is no guarantee that this heuristic returns a feasible solution. However, as motioned above, it is NP-Complete even to find a feasible solution for SPL-PC.

Algorithm 4.3 $GreedyHeuristic(T = (V, A), M, O, E, OR, XOR, D, c, b) \mapsto S^*$

```
1: let r ← root(T)
2: S* ← ∅
3: while Preprocess(T, M, O, E, OR, XOR, D, c, b) and (O ≠ ∅) do
4:     if features(r) is feasible then S* ← features(r)
5:     let Ō ← ∪_{Q∈(OR_r ∪ XOR_r)} Q
6:     if Ō ≠ ∅ then
7:         let v ← argmax_{u∈Ō} b_u/c_u
8:     else
9:         let v ← argmax_{u∈O:(r,u)∈A} b_u/c_u
10:    end if
11:    merge v into r and update T, O, OR, XOR, c_r, b_r, E
12: end while
13: if features(r) is feasible then S* ← features(r)
14: return S*
```

The pseudo-code of the greedy heuristic is shown in Algorithm 4.3. It takes as input the feature tree $T = (V, A)$, the sets M, O, E, OR_v, and XOR_v, for all $v \in V$, as well as the budget D, the cost c_v, and level of importance b_v of each feature $v \in V$. It returns the subset S^* of features with a hopefully feasible product configuration. The root r of the feature tree is identified in line 1. S^* is initialized as empty in line 2, indicating that no feasible solution is known so far. We note that making $S^* = features(r)$ does not necessarily leads to a feasible solution, because it might not satisfy the feature-tree constraints (5) and (6) or the cross-tree constraints (7). The loop in lines 3 to 12 is repeated while there are optional features in the preprocessed feature tree. First, the best known solution is updated in line 4. Let $\overline{O}$ be the set of all exclusive and non-exclusive alternative sets rooted in r (line 5). Next, if $\overline{O}$ is not empty (line 6), then the alternative feature $v \in \overline{O}$ with the best cost benefit ratio is selected in line 7. Features in $\overline{O}$ are prioritized in order to select first features that satisfy the feature-tree constraints (5) and (6). If $\overline{O} = \emptyset$, the sub-feature $v \in O$ of the root r with the best cost benefit ratio is selected in line 9. Feature v is merged with

the root in order to guarantee it is selected, and T, O, OR, XOR, c_r, b_r and E are updated in line 11, the same way as they were in line 16 of Algorithm 4.1. Finally, the best known solution is updated in line 13 and returned it in line 14.

Chapter 5

Computational Results

This chapter reports the results of empirical studies carried out to evaluate the algorithms proposed in Chapter 4. The experiments were conducted on an Intel Core 2 Duo E7500 machine running with a dual-core 2.93 GHz processor and 3 GB of RAM memory and 3MB of cache. Moreover, the algorithms were implemented in Java and compiled with gnu gcc, version 4.4.3. Section 5.1 provides details of the benchmark instances used in the study. Section 5.2 evaluates how much the preprocessing algorithm reduce the size of the feature models. Section 5.3 analyzes the time spent by the backtracking algorithm to find the optimal solution with and without the preprocessing algorithm. Section 5.4 measures the performance of the greedy heuristic, regarding execution time and solution quality. Section 5.5 discussed the threat to validity of the obtained results. Finally, concluding remarks are discussed in Section 5.6.

5.1 Benchmark Instances

We reviewed a large number of research works in the field of SPL and selected a set of ten feature models that were used as benchmark instances. The search was conducted in order to select literature instances of different sizes with and without cross-tree constraints. Table 5.1 shows the feature models sorted by their number of features. Each row depicts a feature model. The first column identifies the feature model (FM). The second column describes the feature model name and provides a reference to the work where the model was introduced (Feature Model [Ref.]). The third and fourth columns describe the number of features ($|V|$) and cross-tree constraints ($|E|$) for each model, respectively. The fifth and sixth columns show the number of mandatory ($|M|$) and optional ($|O|$) features, respectively. Note that half of the feature models have cross-tree constraints. The seventh and eighth columns show the

number of exclusive ($\#xor$) and non-exclusive ($\#or$) alternative features, respectively. Additionally, the ninth column displays the height of the feature tree. Finally, the tenth column depicts the number of valid product configurations ($\#conf$) that satisfies the feature-tree constraints (3) to (6), and the cross-tree constraints (7). This number is provided by the SPLOT tool [Mendonça et al., 2009], and it stops counting when the number is larger then 10^9. This number is greater than the number of feasible solutions for SPL-PC because in SPL-PC the budget constraints must also be satisfied. Additionally, in order to conduct the tests, we generated randomly the values of O, c_v and b_v for all $v \in V$. Note that the goal of this study was to compare between the exact and heuristic algorithms, so we do not worry about getting a feature model instance with more than 213 features, since for 63 features the exact algorithm does not find the optimal solution in less than 2hs. All these instances can be downloaded from Pereira et al. [2014]. In the next section, we aim to analyze the impact of the preprocessing for different types of features and model structures.

Table 5.1. Characteristics of the Original Feature Model Instances

FM	Feature Model [Ref.]	|V|	|E|	|M|	|O|	#xor	#or	height	#conf
#1	Mobile Media [Figueiredo et al., 2008]	17	1	8	9	1	1	4	126
#2	Email System [Thüm et al., 2014]	23	0	3	20	1	0	3	5632
#3	Smart Home [Alferez et al., 2009]	28	3	3	25	1	1	4	182280
#4	Devolution [Thüm et al., 2014]	32	0	11	21	1	4	6	19656
#5	Gasparc [Aranega et al., 2012]	38	0	23	15	4	1	6	352
#6	Web Portal [Mendonça et al., 2008]	43	6	9	34	3	3	5	2120800
#7	FraSCAti [Seinturier et al., 2012]	63	28	19	44	2	0	5	$> 10^9$
#8	Model Transformation [Czarnecki and Helsen, 2003]	89	0	19	70	11	14	8	$> 10^9$
#9	Battle of Tanks [Thüm and Benduhn, 2011]	144	0	8	136	9	1	4	$> 10^9$
#10	e-Shop [Lau, 2006]	213	32	74	139	0	43	8	$> 10^9$

5.2 Preprocessing Algorithm

In this section, we evaluate how much the preprocessing algorithm reduces the size of the instances. Table 5.2 shows the percentage of reduction for each instance. In addition, the last column shows the average time in milliseconds spent by this algorithm.

Table 5.2. Characteristics of the Preprocessed Feature Model Instances

FM	|V|	|E|	|M|	|O|	#xor	#or	height	#conf	Time (ms)
#1	47%	0%	88%	11%	0%	0%	50%	50%	0.04
#2	30%	0%	67%	25%	0%	0%	0%	45%	0.03
#3	18%	33%	67%	12%	0%	0%	0%	82%	0.16
#4	34%	0%	91%	5%	0%	25%	33%	0%	0.23
#5	63%	0%	96%	13%	0%	0%	50%	64%	0.09
#6	19%	0%	89%	0%	0%	0%	0%	0%	0.24
#7	29%	0%	95%	0%	0%	0%	20%	0%	0.38
#8	63%	0%	95%	54%	73%	50%	63%	$> 97\%$	0.11
#9	34%	0%	88%	31%	22%	0%	25%	-	0.19
#10	54%	13%	99%	31%	0%	16%	25%	-	0.69

It can be observed from Table 5.2 that after preprocessing the number of features ($|V|$) in the instances reduces up to 63%. In the worst case, 18% of the features were preprocessed. The cross-tree constraints ($|E|$) were reduced in two out of the five instances with cross-tree constraints. The number of optional features was reduced up to 54%. Furthermore, in seven (out of the ten) instances, the preprocessing algorithm reduces the height of the feature tree. The height is reduced up to 63% (instance #8). Consequently, with the preprocessing of the optional features and change in the models structure, the number of valid product configurations ($\#conf$) is reduced. For example, in the instance #8 the value of $\#conf$ was reduced from more then 10^9 to 30240000. On the other hand, for instances #9 and #10 we know that there was a reduction in the value of $\#conf$ since the number of optional features was reduced, but we could not observed how much reduced, because the number of configurations for these instances is still larger than 10^9. Finally, the average time to preprocess the instances was up to 0.69 milliseconds. In the next section, we show that the preprocessed instances are easier to solve than the original instances.

5.3 Backtracking Algorithm

In this section, we evaluate how much the preprocessing algorithm reduces the execution time of the backtracking procedure. The results of this evaluation are reported in Table 5.3. The first column identifies the instances. The second and fourth columns display the total benefit of the configured product by the backtracking algorithm from original feature model instances and preprocessed feature model instances, respectively. The third and fifth columns show the average time spent in milliseconds for the execution of the backtracking algorithm from original feature models instances and preprocessed feature models instances, respectively. Finally, the sixth column displays the percentage of reduction in the running time when using preprocessed feature model instances. An asterisk indicates instances not solved to optimality before two hours. In this case, the benefit of the best solution found is displayed.

It can be observed in Table 5.3 that the execution time of the backtracking algorithm on the preprocessed instance is always smaller than the execution time of the same algorithm from the original instances. Note that, for the original instance #8 the backtracking algorithm did not find the optimal solution before two hours. On the other hand, using the preprocessing procedure, it takes 183.11 milliseconds to solve this instance. Moreover, for three out of the four original instances where the backtracking cannot find the optimal solution before two hours, the total benefit found by backtracking algorithm using preprocessed instances, was larger than the total benefit found using original instances.

Table 5.3. Average Time to Obtain the Optimal Solution

FM	Original Instance		Preprocessed Instance		Time
	Total Benefit	**Time (ms)**	**Total Benefit**	**Time (ms)**	**Reduction (%)**
#1	320	4.27	320	2.19	49%
#2	200	0.037	200	0.02	46%
#3	780	4487.27	780	1249.92	72%
#4	840	25.22	840	10.73	57%
#5	460	0.2	460	0.09	55%
#6	1280	317.717	1280	215.84	32%
#7	*1020	$> 7.20\text{x}10^6$	*1320	$> 7.20\text{x}10^6$	-
#8	*560	$> 7.20\text{x}10^6$	620	183.11	> 99%
#9	*1080	$> 7.20\text{x}10^6$	*1080	$> 7.20\text{x}10^6$	-
#10	*2350	$> 7.20\text{x}10^6$	*2420	$> 7.20\text{x}10^6$	-

* Better solution found up to 7.200.000 milliseconds (2hs).

5.4 Greedy Heuristic Algorithm

In this section, we evaluate the performance of the greedy heuristic. The results of this evaluation are displayed in Table 5.4. The first column in this table identifies the instances. The second column shows the benefit of the optimal solution S^* found by the backtracking procedure or the best solution found by the backtracking procedure when the optimal solution was not found. The third column shows the benefit of the solution H provided by the greedy heuristic. The fourth column refers to the optimality gap $((S^* - H)/S^*)$ between the benefit of the optimal solution S^* and the benefit of the solution H provided by the heuristic. Finally, the fifth column refers to the average time in milliseconds spent by the greedy heuristic for each feature model instance. The goal is to evaluate the quality of the results and to verify the gains in terms of performance produced by the greedy heuristic.

Table 5.4. Results for the Greedy Algorithm

FM	S*	H	GAP	Time (ms)
#1	320	320	0%	0.35
#2	200	200	0%	0.21
#3	780	760	3%	1.41
#4	840	840	0%	1.98
#5	460	460	0%	0.81
#6	1280	1280	0%	2.05
#7	*1320	1360	-3%	3.26
#8	620	620	0%	0.93
#9	*1080	1240	-15%	1.63
#10	*2420	2540	-5%	6.07

* Better solution found up to 7.200.000 milliseconds (not optimal solutions).

It can be observed that the heuristic finds the optimal solution for all instances solved to optimality, except to the instance #3. The optimality gap of the heuristic in the latter is only 3%. We note that the gap for the instances #7, #9, and #10 are not optimality gaps, because the optimal solution for these instances could not be found by the backtracking algorithm before two hours. For these three instances, where the optimal solution is not known, the heuristic algorithm always found a better solution in a few seconds than the backtracking algorithm in two hours. Additionally, to instances where the optimal solution is known, the heuristic algorithm was significantly faster than the backtracking algorithm.

5.5 Threats to Validity

A key issue when performing these experiments is the validity of the results. Questions we need to answer include: was the study designed and performed in a sound and controlled manner? to which domain can the results generalize? This section presents the different validity threats related to computational results. We presented how the threats were addressed prior to the experiment to minimize the likelihood of their realization and impact. We discussed the experiment validity with respect to the four groups of common threats to validity: internal validity, external validity, construct validity, and conclusion validity [Wohlin et al., 2012].

External validity concerns the ability to generalize the results to other environments, such as to industry practices [Wohlin et al., 2012]. To minimize threats to external validity, we use feature models acquired from existing publications in SPL community. During the selection of the feature models, we use sufficient size and variety to ensure significance. Moreover, to simulate practical situation as far as possible, we randomly generate the constants: D, c_v and b_v, for all $v \in V$. However, we cannot claim that the generated values can be held true in practice. Moreover, the number of benchmark instances used can be considered threat to external validity. However, we consider that our experiments rely on SPLs in important contexts that represent small and medium SPLs.

Threats to internal validity are influences that have not been considered and it can affect the execution time of all exact and heuristic solutions [Wohlin et al., 2012]. In this sense, a limitation of this study concerns the choice of the machine, the language, or the compiler. In addition, the quality of the coding. However, to avoid these effects, we describe the resources used during the computational experiments in details. We reported the average time (in milliseconds) of the execution of each instance 1000 times, in order to avoid as much interference as possible. Moreover, to eliminate machine dependencies, all tests were performed on the same machine and the same compiler. Every effort has been made to minimize the influence of the environment: platform, coding, compiling, paging, caching, etc.

The decision of which feature models choose to include in the studies can be a threat to construct validity (feature models with structure widely used in practice should be considered). To minimize this threat, we chose feature models with different structure and feature types. Furthermore, we documented the characteristics of the feature models used.

Conclusion validity concerns the relation between the treatments and the outcome of the experiment [Wohlin et al., 2012]. A potential conclusion validity threat is if the

benchmark instances are sufficient to the computational evaluation, since we cannot say that these instances are always used in practice. Therefore, in order to ensure the validity, we have opted to analyze only feature models from real-life SPL documented in literature.

5.6 Concluding Remarks

This chapter showed computational results deriving optimized product configurations in feature models from literature SPLs. The performance was measured using ten feature model instances, sampled from several domains. The results showed that the backtracking algorithm take significantly advantage from the preprocessing procedure. The backtracking algorithm on preprocessed instances was always faster than the same algorithm on the original instances. While the backtracking algorithm to original instances found the optimal solution to six out of the ten instances studied in the literature, the backtracking algorithm on preprocessed instances found the optimal solution to seven out of ten instances tested. However, since finding an optimal solution is NP-hard, our results argued the necessity of using a heuristic algorithm for solving the problem. The heuristic algorithm proposed in Chapter 4 found the optimal solution for all instances where the optimum is known, except the instance #3 (*Smart Home*), where the optimality gap was 3%. Moreover, it found a better solution than the backtracking algorithm for three instances where the backtracking did not find an optimal solution in two hours of execution. In the next chapter, we conclude this dissertation, discuss the limitations of related work, and propose future research directions.

Chapter 6

Conclusion

After decades of research in Software Product Line (SPL), a vast number of automated tools for creating and editing feature models have been proposed in the literature (Section 3.1). However, there is still little support for automated product configuration. A study of the tools has shown that product configuration is made manually through successive steps, by one or more developers, until a final product that matches customer requirements is obtained. Therefore, customer requirements are usually neglected, although product configuration has already been investigated in the literature. Guidance and automated support are needed to increase business efficiency when dealing with many possible combinations in an SPL.

This dissertation presented a search-based software engineering (SBSE) approach to tackle the problem of finding the optimal product configuration that satisfies business and costumer requirements. First, a systematic literature review of SPL management tools highlighted that there is a need for automatic product configuration. Next, we conducted a comparative study to choose one of the existing tools to be extended with the needed feature. Based on the results of this study, we extended the FeatureIDE tool with a decision support system for business that reduces the effort required for product configuration. Next, we modeled the problem of finding the optimal product configuration that maximizes the customer satisfaction as an optimization problem. Three algorithms were proposed: a preprocessing algorithm, backtracking procedure and a greedy heuristic. The preprocessing algorithm follows preprocessing the feature model, and it is used with backtracking procedure and a greedy heuristic. The backtracking and greedy heuristic algorithms are exact and approximation algorithm that can be used to solve the problem. Although the backtracking algorithm has exponential complexity, the greedy heuristic algorithm has polynomial complexity with GAP optimality. Our computational results focus over a dataset of ten SPL instances

documented in literature that explore several characteristics of feature models. The backtracking algorithm on preprocessed instances was always faster than the same algorithm on the original instances. Moreover, we empirically verified with our experiments that the optimality GAP of the heuristic is at most 3%. The heuristic was implemented as part of our decision support system and allows new algorithms to be easily incorporated as further characteristics of feature models are examined. Next, the Section 6.1 summarizes the related works and Section 6.2 points out directions for future work.

6.1 Related Work

This section compares related work to our systematic literature review and comparative study of SPL management tools. It also investigated the state-of-the-art about the use of search-based techniques to address SPL product configuration problems.

SPL Management Tools. Analysis of existing SPL management tools has been performed in previous studies, such as the detailed below. The purpose of these studies of a general way is to facilitate tools selection in the context of SPL. However, our systematic review differs from others studies because it includes a large number of tools, characteristics, and functionalities not addressed by previous studies. Moreover, we complement our systematic review with a comparative study detailed between two tools. Our comparative study involved 56 developers taking an advanced SE course. We do not known of any study that perform a comparative analysis of tools through of concrete experimental data.

Beuche et al. [2004] focus on the problems of requirements management for SPL. The result is a comprehensive analysis of four requirements management tools (Borland CaliberRM, Telelogic DOORS, QA Systems IRqA, and RequiLine of RWTH Achen) in the context of SPL based on practical experiences. This paper is intended to direct the attention of both researchers and tool developers to the current problem of inadequate requirements management tools for SPL. Despite our systematic review investigate what are the tools that support requirements management, we do not analyze how this functionality is covered by the tools.

Unphon [2008] compare ten variability modeling and configuration tools (AHEAD, FAMA, Feature Modeling Plug-in, Gears, Kumbang Tools, MetaEdit+, Product Modeler, pure::variants, RequiLine and XFeature). This work categorizes its comparisons into general information (creator, first public release date/year, latest

stable version, cost, and software license), technical infrastructure (application type, implementation language, and supported language), operating systems support (Windows, Mac OS X, Linux, and Unix), rendering of modeling (GUI tree, table, box&arrow, and textual language), format of input/output models support (database, HTML, XML/XMI, CSV, user-define, and grammar), modeling and configuration functionalities (model verification, product family verification, product derivation wizard, and installation wizard), and development functionalities (IDE support, source code generator, link to artefact at requirement level, and link to artifact at design/implementation level). Our systematic literature review complements this study with the assessment of new tools, and different characteristics and functionalities.

Capilla et al. [2007] and Lisboa et al. [2010] aims at finding out how the available tools offer support to the domain engineering process. They defined the current limits of tool support for the processes and they highlighted the need of these tools to incorporate additional functionalities. Capilla et al. [2007] analyze the current state of practice of five SPL management tools, and Lisboa et al. [2010] present an systematic literature review of nineteen SPL management tools. They involved the identification of external characteristics about the tools and of the functionalities they support. On the other hand, our systematic literature review includes not only domain engineering process analysis, but also analysis for the whole domain engineering and application engineering processes. In addition, our study complements these studies with the assessment of new tools and additional characteristics.

The study in Djebbi et al. [2007] was undertaken in collaboration with a group of industries to evaluate three SPL management tools (XFeature, Pure::Variant, and RequiLine). The purpose of the study was two fold: (i) to understand the salient characteristics of SPL management tools, and (ii) to evaluate the ability of existing tools to satisfy the expectations of industries. On the other hand, our comparative study was conducted with two other tools (SPLOT and FeatureIDE).

Another study related to ours was conducted by Simmonds et al. [2011]. This study investigates the appropriateness of different approaches to support variability management. Moreover, it summarizes the usability of eight tools (Clafer, EPF Composer, FaMa-OVM, fmp, Hydra, SPLOT, and VEdit) that support variability management. However, these tools are evaluated by the authors of the paper, while our comparative study is based on concrete experimental data.

SPL Product Configuration. Automated reasoning is an ever challenging field in SPL engineering [Wesowski, 2004]. Botterweck et al. [2007], Mendonça et al. [2009], and Thüm et al. [2014] have proposed visualization techniques for representing staged

feature configuration in SPLs. However, industrial-sized feature models with hundreds or thousands of features make a manual feature selection process hard. Moreover, these approaches focus more on functional features of a product and their dependencies and less on non-functional features. Although it is accepted that in an SPL it is necessary to deal with non-functional features [Kang et al., 1990, 1998], there is no consensus about how to deal with them.

Van Deursen and Klint [2002]; Mannion [2002] investigate automatic analysis of feature models. Van Deursen and Klint [2002] explore automated manipulation of feature descriptions providing algebra to operate on the feature models. Mannion [2002] uses first-order logic for product line reasoning. However this work only provides a model based on propositional logic using AND, OR and XOR logical operators to model SPLs. Both attempts have at least two limitations: (i) they only consider functional features and (ii) they basically aim to answer the single question of how many products a model has.

Benavides et al. [2005] investigated the product configuration problem and applied Constraint Satisfaction Problems (CSPs) to support feature selection. However, they considered that non-functional features only apply to leaf features. In addition, their experiments were conducted in only four SPLs. As a result, the experimentation has shown the exponential behavior of the solution space when the number of features increases. Their implementation has good performance for feature models with up to 25 features. On the other hand, our study (i) considered that the non-functional features are attributes of all features in the feature model; (ii) we have measured the solving time for ten literature SPLs; (iii) our exact algorithm has a good performance up to 43 features (when we use our exact algorithm on preprocessed instances it was able to solve 89 features); (iv) we implemented a heuristic to solve the problem; and (v) our solutions proposed considered cross-tree constraints.

Karatas et al. [2010] used extended feature models to represent non-functional features and introduced a mapping from extended feature models to constraint logic programming over finite domains. Their technique works well for small-scale problems, while for large-scale problems, their technique is too computationally demanding. In contrast, our heuristic algorithm has polynomial complexity that can handle non-functional features and thus can solve the SPL product configuration optimization even on large-scale problems.

Exact techniques show exponential time complexity and do not scale to large industrial feature models with hundreds or thousands of features. However, to the best of our knowledge, there is only one work [White et al., 2009] providing a polynomial complexity algorithm for the optimization product configuration in SPL that adheres

to a set of non-functional features. They proposed the Filtered Cartesian Flattening (FCF) technique. However, their approach still requires significant computing time for large-scale problems. Moreover, this work does not involve feature models instances with cross-tree constraints, their approach only considered attributes in leaf features, and the feature models instances are generated randomly.

Although automated support for product configuration has already been extensively investigated in the literature, customer preferences are usually neglected. We have known of only one study [Bagheri et al., 2010] that represent and consider the customer requirements as main during the product configuration process. Bagheri et al. [2010] present a semi-automated approach to feature model configuration based on a fuzzy propositional language. It uses a variant of propositional logic along with fuzzy logic to represent feature models and their quality attributes, in order to capture both hard and soft constraints of the customers. Our work also considers the customer requirements. However, while this work considers hard and soft constraints of the customer, our work considers the level of importance (none, low, medium, high, very high) of each feature for the customers. Moreover, our study differs from others by providing a decision support system for business.

6.2 Future Work

As future work, we recommend the extension of our exact and heuristic algorithms to include other non-functional features [Kang et al., 1998]. For example, we encourage researchers to extend these algorithms to consider the difficulties of developing and maintaining of the product for the industries, in order to minimize the development effort integrating features to compose a product. Moreover, to simulate practical situation as far as possible, further work should address case studies in industry, in order to ensure that realistic instances of requirements are being generated, through direct contact with the business and customers. Ultimately, our goal was to inspire other researchers to examine further properties in SPL that can lead to the development of new algorithms and eventually to integrate these algorithms with the algorithms introduced in this dissertation.

Bibliography

Alferez, M., Santos, J., Moreira, A.and Garcia, A., Kulesza, U., Araujo, J., and Amaral, V. (2009). Multi-view composition language for software product line requirements. In *2nd International Conference on Software Language Engineering (SLE)*, pages 136–154.

Apel, S., Batory, D., Kästner, C., and Saake, G. (2013). *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer-Verlag.

Aranega, V., Etien, A., and Mosser, S. (2012). Using feature model to build model transformation chains. In *15th international conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 562–578.

Asikainen, T., Männistö, T., and Soininen, T. (2004). Using a configurator for modelling and configuring software product lines based on feature models. In *Workshop on Software Variability Management for Product Derivation, Software Product Line Conference (SPLC)*, pages 24–35.

Bagheri, E., Di Noia, T., Ragone, A., and Gasevic, D. (2010). Configuring software product line feature models based on stakeholders' soft and hard requirements. In *14th International Software Product Line Conference (SPLC)*, pages 16–31.

Baker, P., Harman, M., Steinhofel, K., and Skaliotis, A. (2006). Search based approaches to component selection and prioritization for the next release problem. In *22nd International Conference on Software Maintenance (ICSM)*, pages 176–185.

Barbeau, M. and Bordeleau, F. (2002). A protocol stack development tool using generative programming. *Generative Programming and Component Engineering (GPCE)*, 2487(0):93–109.

Basili, V., Shull, F., and Lanubile, F. (1999). Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(4):456–473.

Batory, D. S. (2005). Feature models, grammars and propositional formulas. In *9th International Software Product Lines Conference (SPLC)*, pages 7–20.

Bednasch, T., Endler, C., and Lang, M. (2003). Captain feature tool. Tool available on SourceForge at: `https://sourceforge.net/projects/captainfeature/` [Online; 01-April-2013].

Benavides, D., Martin-Arroyo, P. T., and Cortes, A. R. (2005). Automated reasoning on feature models. In *17th Conference on Advanced Information Systems Engineering (CAiSE)*, pages 491–503.

Benavides, D., Segura, S., Trinidad, P., and Ruiz-cortés, A. (2007). Fama: Tooling a framework for the automated analysis of feature models. In *1th International Workshop on Variability Modelling of Software Intensive Systems (VAMOS)*, pages 129–134.

Bernardo, M., Ciancarini, P., and Donatiello, L. (2002). Architecting families of software systems with process algebras. *ACM Transactions on Software Engineering and Methodology*, 11(4):386–426.

Beuche, D., Papajewski, H., and Schröder-Preikschat, W. (2004). Variability management with feature models. *Journal Science of Computer Programming*, 53(3):333–352.

Bonifácio, R., Teixeira, L., and Borba, P. (2009). Hephaestus: A tool for managing spl variabilities. In *Brazilian Symposium on Components, Architectures and Reuse Software (SBCARS)*, pages 26–34.

Botterweck, G., Janota, M., and Schneeweiss, D. (2009). A design of a configurable feature model configurator. In *3th International Workshop on Variability Modelling of Software Intensive Systems (VaMoS)*, pages 165–168.

Botterweck, G., Nestor, D., Preubner, A., Cawley, C., and Thiel, S. (2007). Towards supporting feature configuration by interactive visualization. In *1st International Workshop on visualization in Software Product Line Engineering (ViSPLE)*, pages 125–131.

Braga, R., Werner, C., and Mattoso, M. (1999). Odyssey: A reuse environment based on domain models. In *IEEE Symposium on Application-Specific Systems and Software Engineering and Technology (ASSET)*, pages 50–57.

Bryant, R. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691.

Buhne, S., Lauenroth, K., and Pohl, K. (2005). Modelling requirements variability across product lines. In *13th IEEE International Conference on Requirements Engineering*, pages 41–50.

Capilla, R., Sánchez, A., and Dueñas, J. (2007). An analysis of variability modeling and management tools for product line development. In *Software and Service Variability Management Workshop - Concepts, Models, and Tools*, pages 32–47.

Cawley, C., Nestor, D., Preussner, A., Botterweck, G., and Thiel, S. (2008). Interactive visualisation to support product configuration in software product lines. In *2th International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS)*, pages 7–16.

Chen, L. and Babar, M. A. (2011). A systematic review of evaluation of variability management approaches in software product lines. *Journal Information and Software Technology*, 53(4):344–362.

Cirilo, E., Kulesza, U., and Lucena, C. J. P. (2008). A product derivation tool based on model-driven techniques and annotations. *Journal of Universal Computer Science*, 14(8):1344–1367.

Clements, P. and Northrop, L. (2001). *Software product lines: Practices and patterns*. Addison-Wesley.

Cormen, T., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to algorithms*. Mit Press.

Czarnecki, K. and Eisenecker, U. W. (2000). *Generative programming: Methods, tools, and applications*. Addison-Wesley.

Czarnecki, K. and Helsen, S. (2003). Classification of model transformation approaches. Available at: `http://www.ptidej.net/course/ift6251/fall05/presentations/050914/Czarnecki_Helsen.pdf/`.

Czarnecki, K., Kim, C. H. P., and Kalleberg, K. T. (2006). Feature models are views on ontologies. In *10th International Software Product Lines Conference (SPLC)*, pages 41–51.

Czarnecki, K. and Wasowski, A. (2007). Feature diagrams and logics: There and back again. In *11th International Software Product Lines Conference (SPLC)*, pages 23–34.

Deelstra, S., Sinnema, M., and Bosch, J. (2004). Experiences in software product families: Problems and issues during product derivation. In *3th International Software Product Line Conference (SPLC)*, pages 165–182.

Dehlinger, J., Humphrey, M., Suvorov, L., Padmanabhan, P., and Lutz, R. (2007). Decimal and plfaultcat: From product-line requirements to product-line member software fault trees. In *29th International Conference on Software Engineering (ICSE)*, pages 49–50.

Del Rosso, C. (2006). Reducing internal fragmentation in segregated free lists using genetic algorithms. In *International Workshop on Interdisciplinary Software Engineering Research (WISER)*, pages 57–60.

Dhungana, D., Grünbacher, P., and Rabiser, R. (2007). Decisionking: A flexible and extensible tool for integrated variability modeling. In *1st International Workshop on Variability Modelling of Software-intensive Systems*, pages 119–128.

Dhungana, D., Grünbacher, P., and Rabiser, R. (2011). The dopler meta-tool for decision-oriented variability modeling: A multiple case study. *Journal Automated Software Engineering*, 18(1):77–114.

Djebbi, O., Salinesi, C., and Fanmuy, G. (2007). Industry survey of product lines management tools: Requirements, qualities and open issues. In *15th IEEE International Requirements Engineering Conference (IREC)*, pages 301–306.

Eriksson, M., Morast, H., Börstler, J., and Borg, K. (2005). The pluss toolkit: Extending telelogic doors and ibm-rational rose to support product line use case modeling. In *20th International Conference on Automated Software Engineering (ASE)*, pages 300–304.

Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Filho, F. C., and Dantas, F. (2008). Evolving software product lines with aspects: An empirical study. In *30th International Conference on Software Engineering (ICSE)*, pages 261–270.

Frakes, W. B., Prieto-Diaz, R., and Fox, C. (1997). Dare-cots: A domain analysis support tool. In *17th International Conference of the Chilean Computer Science Society*, pages 73–77.

Gallo, G. and Simeone, B. (1989). On the supermodular knapsack problem. *Journal Mathematical Programming*, 45(1):295–309.

Gheyi, R., Massoni, T., and Borba, P. (2006). A theory for feature models in alloy. In *1th Alloy Workshop*, pages 71–80.

Goedicke, M., Köllmann, C., and Zdun, U. (2004). Designing runtime variation points in product line architectures: Three cases. *Journal Science of Computer Programming*, 53(3):353–380.

Griss, M. L., Favaro, J., and d'Alessandro, M. (1998). Integrating feature modeling with the rseb. In *5th International Conference on Software Reuse (ICSR)*, pages 76–85.

Groher, I. and Weinreich, R. (2013). Supporting variability management in architecture design and implementation. In *46th Hawaii International Conference on System Sciences (HICSS)*, pages 4995–5004.

Harman, M. and Jones, B. F. (2001). Search based software engineering. *Journal Information and Software Technology*, 43(0):833–839.

Harman, M., Mansouri, S. A., and Zhang, Y. (2012). Search-based software engineering: Trends, techniques and applications. *Journal ACM Computing Surveys (CSUR)*, 45(1):1–61.

Heidenreich, F., Kopcsek, J., and Wende, C. (2008). Featuremapper: Mapping features to models. In *30th International Conference on Software Engineering (ICSE)*, pages 943–944.

Hervieu, A., Baudry, B., and Gotlieb, A. (2011). Pacogen: Automatic generation of pairwise test configurations from feature models. In *22nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 120–129.

Jain, A. and Biesiadecki, J. (2006). Yam: A framework for rapid software development. In *Second IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, pages 182–194.

Jarzabek, S., Ong, W. C., and Zhang, H. (2003). Handling variant requirements in domain modeling. *Journal of Systems and Software*, 68(3):171–182.

Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, S. (1990). Feature-oriented domain analysis (foda) feasibility study. Available at: `http://www.sei.cmu.edu/reports/90tr021.pdf/`.

Kang, K., Kim, S., Lee, J., Kim, K., Shin, E., and Huh, M. (1998). Form: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5(1):143–168.

Kang, K. C., Lee, J., and Donohoe, P. (2002). Feature-oriented product line engineering. *IEEE Software*, 19(4):58–65.

Karatas, A., Oguztüzün, H., and Dogru, A. (2010). Mapping extended feature models to constraint logic programming over finite domains. In *14th International Software Product Line Conference (SPLC)*, pages 286–299.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of aspectj. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 327–354.

Kim, K., Kim, H., Ahn, M., Seo, M., Chang, Y., and Kang, K. C. (2006). Asadal: A tool system for co-development of software and test environment based on product line engineering. In *28th international conference on Software engineering (ICSE)*, pages 783–786.

Kitchenham, B., Brereton, O. P., Budgen, D., Turner, M., Bailey, J., and Linkman, S. (2009). Systematic literature reviews in software engineering: A systematic literature review. *Journal Information and Software Technology*, 51(1):7–15.

Kitchenham, B., Pfleeger, S., Pickard, L., Jones, P., Hoaglin, D., El Emam, K., and Rosenberg, J. (2002). Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734.

Krueger, C. (2007). Biglever software gears and the 3-tiered spl methodology. In *22th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 844–845.

Krut, J. R. W. (1993). Integrating 001 tool support in the feature-oriented domain analysis methodology. technical report, software engineering institute (sei). Paper available at: `http://repository.cmu.edu/cgi/viewcontent.cgi?article=1166&context=sei`. [Online; 03-June-2013].

Laguna, M. and Hernández, C. (2010). A software product line approach for e-commerce systems. In *7th International Conference on e-Business Engineering (ICEBE)*, pages 230–235.

Lakhotia, K., Tillmann, N., Harman, M., and De Halleux, J. (2010). Flopsy: Search-based floating point constraint solving for symbolic execution. In *22nd International Conference on Testing Software and Systems (ICTSS)*, pages 142–157.

Lau, S. Q. (2006). *Domain analysis of e-commerce systems using feature-based model templates*. PhD thesis, Master's thesis, Dept. Electrical and Computer Engineering, University of Waterloo, Canada. Available at: `http://http://gsd.uwaterloo.ca/node/98`.

Lisboa, L. B., Garcia, V. C.and LucrÃ©dio, D., Almeida, E. S., Meira, S. R. L., and Fortes, R. P. M. (2010). A systematic review of domain analysis tools. *Journal Information and Software Technology*, 52(1):1–13.

Loesch, F. and Ploedereder, E. (2007). Optimization of variability in software product lines. In *11th International Software Product Line Conference (SPLC)*, pages 151–162.

Lotufo, R., She, S., Berger, T., Czarnecki, K., and Wasowski, A. (2010). Evolution of the linux kernel variability model. In *14th International Software Product Line Conference (SPLC)*, pages 136–150.

Mannion, M. (2002). Using first-order logic for product line model validation. In *Second Software Product Line Conference (SPLC)*, pages 176–187.

Mazo, R., Salinesi, C., and Diaz, D. (2012). Variamos: A tool for product line driven systems engineering with a constraint based approach. In *24th International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 1–8.

Mendonça, M., Bartolomei, T., and Donald, C. (2008). Decision-making coordination in collaborative product configuration. In *23rd Annual ACM Symposium on Applied Computing (SAC)*, pages 108–113.

Mendonça, M., Branco, M., and Cowan, D. (2009). S.p.l.o.t.: Software product lines online tools. In *24th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 761–762.

Myllärniemi, V., Raatikainen, M., and Männistö, T. (2007). T.: Kumbang tools. In *11th Software Product Line Conference (SPLC)*, pages 135–136.

Ondrej, R. and Alessandro, P. (2005). Xfeature modeling tool. Available at: `http://www.pnp-software.com/XFeature/`. Automatic Control Laboratory, ETH Zürich [Online; accessed 01-April-2013].

Park, J., Moon, M., and Yeom, K. (2004). Dream: Domain requirement asset manager in product lines. In *International Symposium on Future Software Technology (ISFST)*.

Park, K., Ryu, D., and Baik, J. (2012). An integrated software management tool for adopting software product lines. In *11th International Conference on Computer and Information Science (ICIS)*, pages 553–558.

Pereira, J., Maciel, L., Noronha, T., and Figueiredo, E. (2014). Instances: Search-based product configuration in software product lines. Available at: `http://homepages.dcc.ufmg.br/~juliana.pereira/spl_study`. [Online; accessed 04-January-2014].

Pereira, J., Souza, C. G., Figueiredo, E., Abilio, R., Vale, G., and Costa, H. (2013). Software variability management: An exploratory study with two feature modeling tools. In *Brazilian Symposium on Components, Architectures and Reuse Software (SBCARS)*, pages 20–29.

Pohl, K. (2003). Varmod-prime tool environment. Available at: `http://www.sse.uni-due.de/en/component/content/article/87-forschung/abgeschlossene-projekte/148-varmod-prime-tool-environment`. [Online; accessed 01-April-2013].

Pohl, K., Böckle, G., and Van der Linden, F. J. (2005). *Software product line engineering: Foundations, principles and techniques.* Springer-Verlag.

Prehofer, C. (2001). Feature-oriented programming: A new way of object composition. *Concurrency and Computation: Practice and Experience*, 13(6):465–501.

Salazar, J. R. (2009). *Herramienta para el modelado y configuración de modelos de características.* PhD thesis, MsC Thesis, Dpto. Lenguajes y Ciencias de la Computación, Universidad de Málaga. Available at: `http://caosd.lcc.uma.es/spl/hydra/documents/PFC_JRSalazar.pdf` [Online; accessed 03-January-2014].

Schmid, K., Krennrich, K., and Eisenbarth, M. (2006). Requirements management for product lines: Extending professional tools. In *10th International Software Product Line Conference (SPLC)*, pages 113–122.

Seinturier, L., Merle, P., Rouvoy, R., Romero, D.and Schiavoni, V., and Stefani, J. (2012). A component-based middleware platform for reconfigurable service-oriented architectures. *Software Practice and Experience*, 42(5):559–583.

Simmonds, J., Bastarrica, M., Silvestre, L., and Quispe, A. (2011). Analyzing methodologies and tools for specifying variability in software processes. computer science department, universidad de chile, santiago, chile. Paper available at: `http://www.dcc.uchile.cl/TR/2011/TR_DCC-20111104-012.pdf`. [Online; 03-January-2014].

Sinnema, M. and Deelstra, S. (2007). Classifying variability modeling techniques. *Journal Information and Software Technology*, 49(7):717–739.

Sinnema, M., Deelstra, S., Nijhuis, J., and Bosch, J. (2004). Covamof: A framework for modeling variability in software product families. In *3th International Software Product Line Conference (SPLC)*, pages 197–213.

Spinczyk, O. and Beuche, D. (2004). Modeling and building software product lines with eclipse. In *19th conference on Object-oriented programming systems, languages, and applications (OOPSLA)*, pages 18–19.

Succi, G., Yip, J., and Pedrycz, W. (2001). Holmes: an intelligent system to support software product line development. In *23rd International Conference on Software Engineering (ICSE)*, pages 829–830.

Thao, C., Munson, E. V., and Nguyen, T. N. (2008). Software configuration management for product derivation in software product families. In *15th International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, pages 265–274.

Thüm, T. and Benduhn, F. (2011). Spl2go: An online repository for open-source software product lines. Available at: `http://spl2go.cs.ovgu.de/projects`. [Online; accessed 10-December-2013].

Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., and Leich, T. (2014). Featureide: An extensible framework for feature-oriented software development. *Journal Science of Computer Programming*, 79(0):70–85.

Thurimella, A. K. and Bruegge, B. (2012). Issue-based variability management information and software technology. *Journal Information and Software Technology*, 54(9):933–950.

Thurimella, A. K. and Janzen, D. (2011). Metadoc feature modeler: A plug-in for ibm rational doors. In *15th International Software Product Line Conference (SPLC)*, pages 313–322.

Tracz, W. (1995). A dssa domain analysis tool. *ACM SIGSOFT Software Engineering*, 20(3):49–62.

Travassos, G. and Biolchini, J. (2007). Systematic review applied to software engineering. In *Brazilian Symposium on Software Engineering (SBES)*, pages 1–436.

Unphon, H. (2008). A comparison of variability modeling and configuration tools for product line architecture. Paper available at: `http://www.itu.dk/people/unphon/technical_notes/CVC_v2008-06-30.pdf`. [Online; accessed 15-January-2013].

van der Linden, F., Schmid, K., and Rommes, E. (2007). *Software product lines in action: The best industrial practice in product line engineering*. Springer-Verlag.

Van Deursen, A. and Klint, P. (2002). Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17.

Wesowski, A. (2004). Automatic generation of program families by model restrictions. In *Third Software Product Line Conference (SPLC)*, pages 73–89.

White, J., Dougherty, B., and Schmidt, D. (2009). Selecting highly optimal architectural feature sets with filtered cartesian flattening. *Journal of Systems and Software*, 89(8):1268–1284.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers.

Yoo, S., Harman, M., and Ur, S. (2011). Highly scalable multi-objective test suite minimization using graphics card. In *3th international conference on Search based software engineering (SSBSE)*, pages 219–236.