Supporting Clone-and-Own in software product line

Eddy Ghabach

▶ **To cite this version:**

Eddy Ghabach. Supporting Clone-and-Own in software product line. Software Engineering [cs.SE]. COMUE Université Côte d'Azur (2015 - 2019), 2018. English. NNT : 2018AZUR4056 . tel-01931217

HAL Id: tel-01931217
https://theses.hal.science/tel-01931217

Submitted on 22 Nov 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ÉCOLE DOCTORALE

SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

DOCTORAL THESIS

Supporting Clone-and-Own in Software Product Line

Eddy GHABACH

Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis (i3s)

Presented for the purpose of obtaining a doctor's degree in computer science **from** Université Côte d'Azur
Supervised by: Mireille Blay-Fornarino
Co-supervised by: Franjieh El Khoury, Badih Baz
Submitted on: July 11th, 2018

In front of the jury, composed of:
Jury president:
Philippe Lahire, Professor, Université Côte d'Azur
Reporters:
Abdelhak-Djamel Seriai, Maître de conférences, HDR, Université de Montpellier
Tewfik Ziadi, Maître de conférences, HDR, Campus Pierre et Marie Curie, Université Sorbonne
Examiner:
Laurence Duchien, Professor, Université de Lille
Supervisor:
Mireille Blay-Fornarino, Professor, Université Côte d'Azur
Co-supervisor:
Franjieh El Khoury, Associated member, Laboratoire Eric

ÉCOLE DOCTORALE

SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

Prise en charge du « copie et appropriation » dans les lignes de produits logiciels

Eddy GHABACH

Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis (i3s)

Présentée en vue de l'obtention
du grade de docteur en informatique
d'Université Côte d'Azur
Dirigée par : Mireille Blay-Fornarino
Co-encadrée par : Franjieh El Khoury, Badih Baz
Soutenue le : 11 Juillet 2018

Devant le jury, composé de :
Président du jury :
Philippe Lahire, Professeur, Université Côte d'Azur
Rapporteurs :
Abdelhak-Djamel Seriai, Maître de conférences, HDR, Université de Montpellier
Tewfik Ziadi, Maître de conférences, HDR, Campus Pierre et Marie Curie, Université Sorbonne
Examinatrice :
Laurence Duchien, Professeur, Université de Lille
Directrice de thèse :
Mireille Blay-Fornarino, Professeur, Université Côte d'Azur
Co-encadrante de thèse :
Franjieh El Khoury, Membre associé, Laboratoire Eric

ABSTRACT

A Software Product Line (SPL) manages commonalities and variability of a related software products family. This approach is characterized by a systematic reuse that reduces development cost and time to market and increases software quality. However, building an SPL requires an initial expensive investment. Therefore, organizations that are not able to deal with such an up-front investment, tend to develop a family of software products using simple and intuitive practices. Clone-and-Own (C&O) is an approach adopted widely by software developers to construct new product variants from existing ones. However, the efficiency of this practice degrades proportionally to the growth of the family of products in concern, that becomes difficult to manage. In this dissertation, we propose a hybrid approach that utilizes both SPL and C&O to develop and evolve a family of software products. An automatic mechanism of identification of the correspondences between the features of the products and the software artifacts, allows the migration of the product variants developed in C&O in an SPL. The originality of this work is then to help the derivation of new products by proposing different scenarios of C&O operations to be performed to derive a new product from the required features. The developer can then reduce these possibilities by expressing her preferences (e.g. products, artifacts) and using the proposed cost estimations on the operations. We realized our approach by developing SUCCEED, a framework for SUpporting Clone-and-own with Cost-EstimatEd Derivation. We validate our works on a case study of families of web portals.

Keywords: SUCCEED, Software Product Line Engineering, Software Product Line Evolution, Software Product Variants, Software Reuse, Software Derivation, Clone-and-Own, Feature Location, Feature Model, Software Variability.

RESUMÉ

Une Ligne de Produits Logiciels (LPL) supporte la gestion d'une famille de logiciels. Cette approche se caractérise par une réutilisation systématique des artefacts communs qui réduit le coût et le temps de mise sur le marché et augmente la qualité des logiciels. Cependant, une LPL exige un investissement initial coûteux. Certaines organisations qui ne peuvent pas faire face à un tel investissement, utilisent le "Clone-and-own" (C&O) pour construire et faire évoluer des familles de logiciels. Cependant, l'efficacité de cette pratique se dégrade proportionnellement à la croissance de la famille de produits, qui devient difficile à maintenir. Dans cette thèse, nous proposons une approche hybride qui utilise à la fois une LPL et l'approche C&O pour faire évoluer une famille de produits logiciels. Un mécanisme automatique d'identification des correspondances entre les "features" caractérisant les produits et les artéfacts logiciels, permet la migration des variantes de produits développées en C&O dans une LPL. L'originalité de ce travail est alors d'aider à la dérivation de nouveaux produits en proposant différents scenarii d'opérations C&O à effectuer pour dériver un nouveau produit à partir des features requis. Le développeur peut alors réduire ces possibilités en exprimant ses préférences (e.g. produits, artefacts) et en utilisant les estimations de coûts sur les opérations que nous proposons. Les nouveaux produits ainsi construits sont alors facilement intégrés dans la LPL. Nous avons étayé cette thèse en développant le framework SUCCEED (SUpporting Clone-and-own with Cost-EstimatEd Derivation) et l'avons appliqué à une étude de cas sur des familles de portails web.

Mots clés: SUCCEED, Ingénierie des Lignes de Produits Logiciels, Evolution des Lignes de Produits Logiciels, Variantes de Produits Logiciels, Réutilisation de Logiciels, Dérivation de Logiciels, Clone-and-Own, Identification des Caractéristiques, Diagramme de Caractéristiques, Variabilité Logicielle.

ACKNOWLEDGEMENTS

Thank you ***God***, you are the main source of my energy and reason of my success.

Thanks to my ***family***, lovely *parents* and supporting *brother*, who all missed me during my journeys in France. For sure, I couldn't make it without you. Thanks for your endless support.

Thanks for the opportunity that gave me the *French state* to do my research on its land. I am grateful to my research laboratory members *I3S*, and especially to the *SPARKS* team to which I belong. A big thank you to ***Mireille Blay-Fornarino***, who supervised my thesis during the past years. I appreciate so much her patience, especially during the first period, since it was really difficult to work together remotely. I would like to thank her for her continuous encouragement and her enthusiasm, while she was supporting me with all required knowledge and time to progress in my research, essentially during my journeys in France. It was a great honor for me to work with her, and I was lucky to work with someone who has that much of experience and knowledge in the research domain I chosen. I also thank my colleagues, *Cécile Camilleri* who gave me access to *Rockflows* framework, *Philippe Collet* and *Sebastien Mosser* who provided me with theirs advises.

Thanks to *Université Saint-Esprit de Kaslik (USEK)*, my university in Lebanon, for facilitating my research mission and providing me with the necessary academic resources and access to the digital library during my research period. Thanks to my co-supervisors, ***Franjieh El Khoury***, who supported me with her guidance and knowledge, reviewed my work, and encouraged me throughout my research period, and ***Badih Baz***, who supported me and believed in my capabilities.

Thanks for the *National Center for Scientific Research in Lebanon (CNRS-L)* and the *PHC Cedre Program* doctoral scholarships.

Thanks to the *jury members* for their precious opinions and for everyone reading this dissertation.

لا تَكُن حَبَّةَ مِلحٍ تَذوبُ في المَاء
إنَّما نُقطَةَ زَيتٍ تَسطَعُ بَهاء

آدي غَبَش

Don’t be a grain of salt that dissolves in water,
but a spot of oil that shines brightly

Eddy Ghabach

Ne soyez pas un grain de sel qui se dissout dans l’eau,
mais une tache d’huile qui brille vivement

Eddy Ghabach

TABLE OF CONTENTS

1 Introduction 1
- 1.1 Context and Motivation . . . 2
- 1.2 Running Example . . . 3
- 1.3 Challenges . . . 4
- 1.4 Contributions . . . 7
- 1.5 Organization of the Dissertation . . . 8

I Background and State of the Art 11

2 Background 13
- 2.1 Software Reuse . . . 14
- 2.2 Clone-and-Own Approach . . . 16
- 2.3 Software Product Lines . . . 17
- 2.4 Summary and Contribution Decisions . . . 28

3 Related Work 31
- 3.1 Software Product Line Adoption . . . 32
- 3.2 Product Derivation Support . . . 41
- 3.3 Software Product Line Evolution . . . 45
- 3.4 Summary and Contribution Choices . . . 48

II Approach Contributions 51

4 Migration process 55
- 4.1 Introduction . . . 56
- 4.2 Product Line Definition . . . 58
- 4.3 Correlations Identification . . . 63
- 4.4 Product Line Validation . . . 68
- 4.5 Product Line Limitations . . . 69
- 4.6 Summary . . . 71

5 Configuration Process 73
- 5.1 Introduction . . . 74
- 5.2 Configuration . . . 75
- 5.3 Configuration Modes . . . 78
- 5.4 Configuration Scenarios . . . 79
- 5.5 Derivation Operations . . . 83

5.6 Summary 87

6 Towards Cost-Estimated Derivation **89**
6.1 Introduction 90
6.2 Cost-Estimation 90
6.3 Summary 96

7 Derivation and Evolution Process **97**
7.1 Introduction 98
7.2 Product Derivation 99
7.3 Product Line Evolution 102
7.4 Summary 108

III Implementation and Validation **109**

8 SUCCEED Framework **111**
8.1 Introduction 112
8.2 Migration Process 112
8.3 Configuration Process 115
8.4 Derivation Process 117
8.5 Evolution Process 119
8.6 Summary 120

9 Approach Validation **121**
9.1 Introduction 122
9.2 Validation 122
9.3 Limitations 125
9.4 Threats to Validity 125
9.5 Summary 126

IV Conclusion and Perspectives **133**

10 Conclusion and Perspectives **135**
10.1 Conclusion 136
10.2 Perspectives 138

List of Abbreviations **141**

List of Figures **143**

List of Tables **145**

List of Algorithms **147**

Table of Objectives **149**

Table of Definitions and Properties **151**

Table of Examples 153

Table of Listings 155

Bibliography 157

CHAPTER 1

INTRODUCTION

Contents

1.1 Context and Motivation 2
1.2 Running Example 3
1.3 Challenges 4
1.4 Contributions 7
1.5 Organization of the Dissertation 8

1.1 Context and Motivation

In software industry, many are the organizations that develop a family of software products for a group of customers, that belong to the same market segment. These organizations vary in size, in terms of staff, intellectual and financial resources, from start-ups and small organizations to large enterprises [TH03].

In general, large enterprises study and identify their market segment and product portfolio, as an initial step, before starting the development process [CN01, PBV05]. Similarly to other domains in industry, such as automotive industry, mass customization is adopted by organizations that focus on developing and maintaining a family of software products instead of developing many individual products [Kru01, PBV05]. Therefore, they are able to determine the main features of the family of products to develop, and plan to develop these products in a way that allows their reuse. Some of these organizations adopt Software Product Line Engineering (SPLE) approach, which consists on developing artifacts adaptable in several products in a *domain engineering* process, before deriving the products in an *application engineering* process by exploiting the developed artifacts [WL99, PBV05, DSB05, LSR07, ACR09]. A Software Product Line (SPL) is a set of software products that belong to the same domain and have some characteristics in common [CN01]. These characteristics are known as *features* [BLR+15]. A Feature Model (FM) is one of the abstract representations of SPL products variability [KCH+90]. A *configuration* is a selection of features that respects the constraints imposed by the FM and generally reflects a product of the SPL [BEG+11]. SPLs permit a systematic reuse of software artifacts, which reduces development cost and increases time to market and software quality [TH03, PBV05].

Developing the artifacts of an SPL through the domain engineering process, before deriving new products throught an application engineering phase, is considered as a large and expensive up-front investment, that several organizations are not able to afford [PBV05]. Therefore, in practice, most small organizations do not develop an SPL from scratch [AM14], but often, start with developing a successful product, that grows later on into a family of products [BBS11]. For instance, a start-up or a small organization aiming to develop software products, focuses on providing high quality and fast delivered products to its very new customers, in order to position itself on the market and attract more customers. Thus, it concentrates on developing a single product at a time without planning for future products releases. Short-term thinking prevents some organizations from initially predicting that they are going to develop a family of products, and they realize it when customers requirements emerge over time. Consequently, this prevents organizations from investing enough time and resources to support and manage reuse during development process [DRB+13]. Such organizations develop software products by adopting a simple ad-hoc technique such as copy-paste-modify [ZFdSZ12, Mar16], or the well-known Clone-and-Own (C&O) adopted when developing products through a Version Control System (VCS). C&O is an approach that consists in cloning an existing Product Variant (PV) then modifying it to add and/or remove some functionalities in order to obtain a new PV [ZPXZ12, DRB+13, FLLHE14, LBC16]. Due to simplicity, availability and rapidity that it provides, this approach is practically adopted by many organizations as “favorable and natural” solution to develop a family of related software systems [DRB+13]. Although being a time and cost saving practice, C&O might turn into an expensive and inefficient solution if tracking about the artifacts existing in several clones is lacked, which produces an incertitude in identifying the PV(s) to be considered as source for cloning [DRB+13, AM14, LBC16].

A possible alternative is the migration of the existing PVs into an SPL, in order to manage their variability and benefit from a systematic reuse [CN01]. This process is known as extractive [Kru01] or bottom-up [MZB+15b] adoption, or re-engineering [ZHP+14, AM14, ALHL+17] of software product lines. As per Ziadi *et al.*, a manual reverse engineering is error-prone and time-consuming [ZFdSZ12]. Thus, an automated approach is required to integrate the existing PVs into an SPL.

Evolving a family of software products consists often in deriving new variants by reusing the existing ones. Despite that SPLs provide systematic reuse, due to variability management, product derivation is restricted to the product line portfolio. Hence, deriving new products consists of evolving the SPL at both domain and application engineering levels, a task that is considered complex due to variability and interdependency between products [BP14].

Several works in literature have proposed extractive SPL adoption frameworks and approaches to enhance C&O [AmSH+13, RC13a, FLLHE14, MZB+15b]. These frameworks disparately allow the integration of existing PVs, support their systematic reuse and enhance C&O with possible derivation – automated or sometimes assisted with hints – and integration of new PVs. However, these approaches do not provide software engineers with the freedom that C&O offers them to create new PVs. In C&O, during product derivation, software engineers are the decision makers. The "own" is gained when software engineers are aware how the product is constructed, since they decide what and how to clone. The proposed approaches aim to automate the clone and impose their solution on software engineers, that are not able to recognize from which PVs the artifacts of the derived PV where cloned. C&O practitioners consider that any alternative approach, in order to convince them, must offer the advantages provided by C&O such as availability, simplicity and independence [DRB+13].

1.2 Running Example

We illustrate in the upcoming example, three PVs for managing soccer matches[1]. The PVs are web applications implemented using markup (HTML), style sheet (CSS), scripting (JavaScript), and object-oriented (Java classes and servlets) languages.

Table 1.1: Running example product variants with their corresponding features

Product	**Feature**			
	ManageMatches	AddMatches	ModifyMatches	DeleteMatches
p_1	✓	✓	✓	
p_2	✓	✓	✓	✓
p_3	✓	✓		

Table 1.1 shows the business functionalities – a.k.a features – implemented by each variant, and Table 1.2 shows an excerpt of the files – a.k.a. assets – used by each variant to implement the features, and their corresponding versions. Product p_1 allows to *manage*, *add* and *modify* matches. Product p_2 allows to *delete* matches in addition. Product p_3 allows to *manage* and *add* matches only. For simplicity and to make the example comprehensive, we show in Table 1.2 only an excerpt of the assets, and we represent the assets by their names and not their relative path within

[1] The implementation files of the PVs of the running example are available on the following git repository link: https://github.com/eddyghabachi3s/SoccerManager

the projects. For instance, $DeleteMatch.java$ refers to $src/Match/DeleteMatch.java$. Figure 1.1 shows the main interfaces of the 3 variants. We demonstrate our approach on this running example throughout the dissertation.

Table 1.2: Running example product variants with an excerpt of their corresponding assets

Product	Asset^{version}
p_1	match.jsp^1 SaveMatch.java^1 style.css^1
p_2	match.jsp^1 SaveMatch.java^1 style.css^2 DeleteMatch.java^1
p_3	match.jsp^1 SaveMatch.java^2 style.css^3

1.3 Challenges

In this section, we identify the challenges that might face an organization at development level when constructing and evolving a family of software products. In our dissertation, we do not address other aspects such as organizational challenges that were addressed widely by Bosch [Bos10, BBS11].

Challenge 1: Supporting the derivation of new product variants
How to guide software engineers to derive new product variants?

A family of software products is composed of a set of products that are developed to respond to the requirements of a group of customers that belong to the same market segment. Thus, these products share a set of common characteristics, while they differ from each other due to some variable characteristics that are implemented by some products and not by others, and product specific characteristics where each is implemented specifically by a single product of the family [BP14]. Arising customer requirements and technology changes necessitate the development of new product variants. Hence, the derivation of a new product variant is needed whenever the family of software products does not offer a variant that implements all and only the required features. Given the family of products presented in the running example, to deliver a product that allows to *manage*, *add* and *delete* matches, a new product variant – say p_4 – has to be derived, since none of the existing variants implements all and only the requested features.

Challenge 1.a: Mapping features to assets
How to determine which assets contribute in the implementation of each feature?

When the derivation of a new product variant is needed, reusing the artifacts of existing variants is advantageous. To do so, first the features implemented by each existing product

Matches

Home Team	Away Team	Result	Modify
Bayern	Sevilla	0 - 0	Modify
Liverpool	Roma	5 - 2	Modify
Real Madrid	Bayern	2 - 2	Modify
Real Madrid	Juventus	1 - 3	Modify

Add

(a) Product p_1 main interface

Matches

Home Team	Away Team	Result	Modify	Delete
Bayern	Sevilla	0 - 0	Modify	Delete
Liverpool	Roma	5 - 2	Modify	Delete
Real Madrid	Bayern	2 - 2	Modify	Delete
Real Madrid	Juventus	1 - 3	Modify	Delete

Add

(b) Product p_2 main interface

Matches

Home Team	Away Team	Result
Bayern	Sevilla	0 - 0
Liverpool	Roma	5 - 2
Real Madrid	Bayern	2 - 2
Real Madrid	Juventus	1 - 3

Add

(c) Product p_3 main interface

Figure 1.1: Main interfaces of the running example variants

variant have to be identified. For instance, to reuse the feature $DeleteMatches$ to derive p_4, one must know that it is implemented only by product p_2. The implementation of a feature is realized in one or more assets of the products that implement the feature. Therefore, identifying the assets that contribute in the implementation of a certain feature is necessary for reuse in order to determine which assets are required for the derivation of the new product.

Challenge 1.b: Identifying the possible scenarios and operations to achieve the derivation

What are the products or combinations of products that can be a source of reuse to achieve the derivation, and respectively, what are the operations that must be performed on the cloned assets to construct the new product?

A feature required for the derivation of a new product can be implemented by several existing products. Further, the set of the required features can be spread on several products which necessitate the reuse of assets that belong to different products. Thus, determining the combinations of products that implement the set of required features for the derivation is a major task. An existing product might implement some features that are not required for the derivation

of the new product, and therefore, those features must be identified. For instance, a possible scenario to derive p_4 is to clone p_2 and remove from the clone the implementation fragments corresponding to feature $ModifyMatches$. Consequently, the assets of those products that implement the required features might implement also some unrequired features. Hence, for each required asset that has to be cloned, the operation that must be performed on it has to be specified. For example, considering that the asset $SaveMatch.java$ of the product p_2 is mapped to the feature $ModifyMatches$, since the latter is not required, the operation to be performed on the asset $SaveMatch.java$ is to clone it and remove from it the implementation fragments corresponding to $ModifyMatches$.

Challenge 1.c: Facilitating derivation choices
What might be the selection factors and indicators that help software engineers to make the derivation choices? What is the cost to perform each asset level operation? Respectively, which scenario provides the least expensive derivation cost?

Several scenarios might be available to achieve the derivation, for instance, based on what a software engineer can decide to derive p_4 either by cloning p_2 and removing from the clone the code fragments related to $ModifyMatches$, or by cloning p_3 and extracting the code fragments related to $DeleteMatches$ from p_2 and integrate it in the clone. One selection factor could be her preference to work with p_2 (that she is more familiar with) or p_3 (that is the recent derived product) . Further, some of the assets corresponding to the products of a possible scenario may require only to be cloned without being modified since they are mapped only to the required features. Other assets require to be cloned and modified in order to remove implementation fragments corresponding to unrequired features. Moreover, several asset level operations might be identified to construct the asset implementing the required features. For instance, referring to Table 1.2, there exists 3 versions of $style.css$, hence, an operation might consist of removing some features from a version, while another operation might consist of collecting features from several versions. Since several scenarios (combinations of products) and several asset level operations can be identified, the selection of the appropriate scenario and operations might become a difficult task. Therefore, providing valuable indicators to software engineers about the products of each scenario and the assets of each operation can facilitate their derivation choices, since they will be able to construct the derivation scenario based on their own preferences. Moreover, estimating the cost of an asset level operation and respectively the cost of a possible scenario can facilitate the selection of the operations to perform to achieve the derivation.

Challenge 2: Evolving the family of software products
How to integrate the newly derived variants in the family of software products?

The derivation of new products involves the definition of new features and the construction of new assets. Integrating the newly derived products into the family of software products is necessary to allow their reuse in future derivations. Moreover, the artifacts added during derivation can enhance the derivation of more products.

Challenge 2a: Helping to structure features when adding a new variant
How to update the structure of the features of the family of products to integrate the features added during the configuration?

The derivation of new products comes mostly from the need of new features that were not offered by the existing products. Hence, it is important to allow software engineers to simply define those features during the configuration of a new variant. Furthermore, once the variant is integrated into the family of software products, its new features must be integrated as well. Hence, it is necessary to update the structure of the features of the family of products, and redefine the constraints that manage their selection.

Challenge 2b: Managing the addition of new products
How to support the integration of the newly derived products by clone-and-own into the family of products? How to guarantee that the integration of a new product does not prohibit the derivation of existing ones?

It is important to allow a smooth and incremental integration of the newly derived products into the family of software products in order to benefit from their systematic reuse in future derivation and employ their new artifacts to derive new products. Consequently, it is necessary to determine the impact of this integration on the family of products to guarantee that evolving it by integrating new products preserves the derivation of the existing ones as well.

1.4 Contributions

In this dissertation, we address the development of a family of software products based on C&O. We focus on the segment of software engineers that are familiar with C&O and looking for support during the derivation of PVs. Thus, the main goal of this dissertation is to support software engineers in deriving new PVs based on C&O. To fulfill this goal, we propose a hybrid approach that allows on the one hand an automated derivation of existing PVs after migrating them into an SPL, and on the other hand it supports the derivation of new PVs based on C&O by providing the possible operations to perform at asset level, in order to derive the desired product. To facilitate the choice of the operations to perform during product derivation, we define correlation indicators and functions in order to estimate the cost of an operation. Hence, software engineers can rely on their own preferences and the proposed cost estimations in order to achieve the derivation. To enable SPL evolution, our approach permits the integration of the newly derived products in the established SPL. The contributions of our approach are as follows:

1. The first contribution consists of a novel light mechanism to determine mappings between features and assets, while migrating the existing PVs into an SPL. We define those mappings as *correlations*. Correlations serve in facilitating reuse and maintenance, since they allow to determine which assets contribute in the implementation of each feature. Meanwhile, the migration of the existing PVs into an SPL enables also their automated derivation.

2. In the second contribution, we aim to support software engineers in deriving new PVs based on C&O. Instead of imposing a single solution, we propose the possible *configuration scenarios* by means of *operations* to perform at asset level, in order to derive a new product variant. A *configuration scenario* provides a top-down overview, specifying the combination of products to rely on, and the features to retain or remove from them in order to construct the *desired* product. We assign to each configuration scenario, the operations to perform in order to accomplish the derivation. *Operations* consist of the

actions to be made at asset level, such as removing or extracting the implementation fragments corresponding to a certain feature. We auto-generate an FM based on the identified configuration scenarios and operations. The generated FM can be configured either by choosing one of the proposed configuration scenarios, or by selecting for each required asset one of its proposed operations, regardless the configuration scenarios. Moreover, the generated FM allows a software engineer, whenever possible, to prevent operations that correspond to a product or a version of an asset that she is not familiar with.

3. The third contribution consists on providing a *cost estimation* of the operations to perform, and respectively the proposed configuration scenarios. In order to facilitate the choice of an operation or a configuration scenario, we define *correlation indicators and functions* that allow to estimate the cost of a certain operation. Software engineers can rely on the estimated cost as a valuable selection parameter while deriving a new product.

4. The fourth contribution consists of providing a reactive SPL evolution by allowing an incremental integration of the newly derived products in the SPL. This integration coincides with an incremental update of the correlations. The integration of the newly derived products into the SPL permits their automated derivation as well as the existing ones and allows the reuse of their artifacts in future derivations.

We realized our approach by developing a framework for *SUpporting Clone-and-own with Cost-EstimatEd Derivation (SUCCEED)*. *SUCCEED* provides an incremental integration of PVs into an SPL and a guided derivation of new PVs based on the approach that we propose in this dissertation. We evaluated our approach on the incremental derivation of 5 products based on an initial product line of 3 products.

1.5 Organization of the Dissertation

This dissertation is organized into four parts.

Part I presents the literature review and consists of two chapters:

— In **Chapter 2**, we present the background in which we demonstrate the two reuse practices that we rely on to construct our approach which are *Clone-and-Own* and *Software Product Lines*. In this chapter, we point on the necessity of reuse, and we clarify the purpose of the hybrid approach that we adopt, by taking benefits and avoiding drawbacks of each of the two presented practices.

— In **Chapter 3**, we present the related works, and we position our work compared to them. On this basis, we identify our contributions choices.

Part II presents our approach contributions and it consists of four chapters:

— In **Chapter 4**, we provide our definition of an SPL and demonstrate the process of migrating PVs into it. Further, we present the mechanism of identifying correlations.

— In **Chapter 5**, we present the configuration process, and we demonstrate how we identify for each configuration its possible configuration scenarios and operations to perform.

— In **Chapter 6**, we define correlation indicators and functions that we use to estimate the cost of a certain operation, and respectively the cost of a configuration scenario.

— In **Chapter 7**, we present the derivation and evolution process, in which we demonstrate how we support the derivation with a constraints system allowing software engineers to select favorable configuration scenario and operations based on their own preferences and the proposed cost estimation. Furthermore, we demonstrate how a newly derived product can be integrated in the SPL and how correlations and FMs are updated.

Part III presents the *SUCCEED* framework and approach validation:

— In **Chapter 8**, we present SUCCEED, the framework that we developed to implement and test our approach.

— In **Chapter 9**, we evaluate our approach on a case study that consists of a product line of 8 product variants.

Part IV concludes the dissertation:

— In **Chapter 10**, we conclude the dissertation by summarizing its contributions and exposing the perspectives of this work.

Part I

Background and State of the Art

CHAPTER 2

BACKGROUND

Contents

2.1 Software Reuse **14**
2.2 Clone-and-Own Approach **16**
2.2.1 Definition 16
2.2.2 Benefits and Drawbacks 16
2.3 Software Product Lines **17**
2.3.1 Definition 17
2.3.2 Software Product Line Engineering 18
2.3.2.1 Domain Engineering 19
2.3.2.2 Application Engineering 21
2.3.3 Benefits and Drawbacks 21
2.3.4 Variability Management 23
2.3.5 Feature Model 24
2.3.6 Product Configuration 26
2.3.7 Product Derivation 27
2.4 Summary and Contribution Decisions **28**

2.1 Software Reuse

In software industry, same as any domain in business industry, supplier interests contradict with customer interests [TH03]. A supplier aims to deliver a product with the minimum development effort and the maximum profitability, while a customer aims to receive the product in a short period of time, providing a set of required functionalities adapted for her needs. Responding to the intensive market demands, put suppliers in front of challenges that start early at the analysis and design activities in the software development life cycle and do not end after delivering the software product. Those challenges are:

— ***Cost & Effort:*** develop the software product with the lowest cost and minimum effort.

— ***Quality:*** ensure a high software quality in terms of efficiency and effectiveness.

— ***Delivery:*** deliver the software product in a short period of time.

— ***Maintenance:*** provide a quick and smooth maintenance of the software product when needed.

In July 2017, similarily to preceding years, the Gartner Group announces that "enterprise software" spending will have the highest annual growth rate in the IT sector in 2018, with forecasts of 8.6 percent for 2018 [Gro17]. In fact, this indicator reflects the large amount of functionalities offered in today's enterprise software systems, to serve the progressing amount of customer requirements. Standardized mass products could not satisfy the particular requirements of all customers. Therefore, mass customisation became a need to respond to the high demand for individualized products [PBV05].

> “ **Mass customisation** *is the large-scale production of goods tailored to individual customers' needs.*
>
> [DAV87]

Although individualized products differ from each other by providing specific functionalities, they belong to the same market segment, and therefore, have a large portion of functionalities in common. The increase of customer requirements raises up several problems, due to the growth of the software size and respectively the development complexity. These problems can be identified as follows [TH03]:

— A functionality may behave differently in various products.

— A functionality may be redeveloped in various products.

— A change to a functionality in a certain product may not be propagated to the other products implementing it.

— A change to a functionality has to be repeatedly done on the products implementing it.

To cope the challenges and avoid the problems listed above, software developers rely on software reuse defined by Kang as *"the process of implementing new software systems using existing software information"* [KCH+90].

> **Software reuse** *is the process of creating new software by reusing pieces of existing software rather than from scratch.*
>
> [Kru92]

Code reuse is a practice that software developers rely on to save time and efforts by reusing predefined methods, ready-made libraries and off-the-shelf software components. Moreover, reuse can be applied on model components, architecture components, documentation and test cases. In software development, several approaches are adopted to enforce reuse, such as Model-Driven Engineering [Sch06, AR13], Component-Based Software Development [JLL05], Service Oriented Software Engineering [BL07] and Feature Oriented Software Development [KLD02, KA13, ABKS16]. Moreover, ad-hoc reuse practices, such as copy-paste-modify and clone-and-own are adopted by several organizations to derive new product variants [ZPXZ12, DRB+13].

> The **Feature-Oriented Reuse Method** *concentrates on analyzing and modeling a product line's commonalities and differences in terms of features and uses this analysis to develop architectures and components.*
>
> [KLD02]

Objective 1:
Reuse in feature-oriented software development

In this dissertation, we focus on adopting reuse in feature-oriented software development. In this context, practically software reuse is performed either by applying the clone-and-own approach as a convenient and quick solution for light development projects [DRB+13, LBC16] or Software Product Line Engineering as platform-based solution for complex development projects [CN01, TH03, PBV05].

2.2 Clone-and-Own Approach

2.2.1 Definition

Clone-and-own is an approach that consists on cloning an existing product variant, then modifying it to add and/or remove some functionalities, in order to obtain a new product variant [ZPXZ12, DRB+13, FLLHE14, LBC16]. Due to the extensive existence of open source projects and public repositories, clone-and-own became the trending favorite and natural cloning approach adopted by software developers. Although cloning is done at different software development stages, it occurs more often at the code level [DRB+13].

> “ The **Clone-and-Own** *is a common practice in families of software products consisting of reusing code from methods in legacy products in new developments.*
>
> [LBC16]

2.2.2 Benefits and Drawbacks

Considering clone-and-own as a software reuse approach is due to the following benefits that it provides [DRB+13]:

— The *availability of the software variants* that are generally hosted on public remote or local sites.

— The *simplicity of cloning* an existing software variant into a new one ready for modification.

— The *rapidity* to perform the clone and start the modification process.

— The *specified* and *verified functionality* already implemented in the cloned product variant.

— The *freedom* that the software developer benefits from to make any modification, without being dependent on the product variant from which the clone was made.

Although being a time and cost saving approach, clone-and-own confronts several drawbacks [DRB+13]:

— The *lack of information* about the connection between clones.

— The *difficulty in propagating changes* made on a product variant to other product variants, due to the lack of connection information and the inability of automatically determining the product variants that require the change.

— The *repetitive tasks* performed on several product variants, for example, in order to propagate a change, which makes the maintenance process very expensive and complex.

— The *incertitude* in identifying the product variant(s) to be considered as the source for cloning.

— The *lack of reuse tracking* about the artifacts existing in several clones. This information is not consolidated except in the developers mind.

Despite its drawbacks, Dubinsky *et al.* [DRB+13] showed through a study, conducted on six software product lines, and realized via code cloning, that C&O is widely adopted by software developers, who consider it the simplest and natural approach for reuse. They affirm that *"it accelerates development, since they use what they already have; it saves time and, therefore, it saves money"* [DRB+13]. Moreover, in their study about industrial SPLs, Dubinsky *et al.* affirm that, any approach that can be considered as an alternative for cloning has to provide the simplicity and availability provided by cloning.

OBJECTIVE 2:
ADOPTING CLONE-AND-OWN AS A STARTING POINT

Since it is a fact that no alternative approach is more efficient than cloning in the early development stages of a family of software products, and clone-and-own is the predominant approach adopted by software developers, we focus on adopting clone-and-own as the starting point in any solution addressing product variants development.

2.3 Software Product Lines

2.3.1 Definition

Standard software products lack diversification and consequently fail to satisfy customer needs [PBV05]. Taking into account customer requirements by delivering individualized products requires a large investment from the product developer and increases the development cost. Software product lines were invented as a solution that combines platform-based development and mass customization, allowing a reduction of the development cost and a large variety of products [PBV05].

> "A **software product line** *is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.*
>
> [CN01]

Using an SPL approach, a product is not implemented from scratch. Yet, it is integrated in the SPL by exploiting its variability mechanism, reusing components and adding the missing ones [TH03].

> “ **Core assets** *are those reusable artifacts and resources that form the basis for the software product line. Core assets often include, but are not limited to, the architecture, reusable software components, domain models, requirements statements, documentation, specifications, performance models, schedules, budgets, test plans, test cases, work plans, and process descriptions.*
>
> [NCB+07]

2.3.2 Software Product Line Engineering

Software Product Line Engineering (SPLE) was raised on top of a theory born in 1980s based on increasing economy of scale by adopting a constructive reuse of software artifacts.

> “ **Product line engineering** *is a process that delivers reusable components, which can be reused to develop a new application for the domain.*
>
> [TH03]

As per Northrop *et al.* [Nor02, NCB+07], the product line development is composed of three main activities: (as shown in Figure 2.1)

— **Core asset development**: the core asset development activity consists on developing the core assets to be used as ingredients for product derivation. This activity takes as an input the product constraints including the commonalities and variations, the production constraints and strategy defining how the product line is going to be built, and the pre-existing assets. The output of the core asset development activity is the core assets themselves, in addition to the product line scope and the production plan that defines the process to be used to derive the products.

— **Product development**: the product development activity takes as an input the three outputs of the core asset development activity, in addition to the individual requirements of the product to be produced. The derivation of a product can affect the output of the core asset development activity; therefore, this process is iterative. The output of this activity is the products themselves.

— **Management**: the core asset development and product development activities must be accompanied by technical and organizational management. The technical management activity ensures that the core assets development and the product development activities follow the process defined for the product line. The organizational management defines the structure of the company and guarantees that its organizational units are receiving the required resources to build the product line.

> The **product line scope** *is a description of the products that will constitute the product line or that the product line is capable of including.*
>
> [NCB+07]

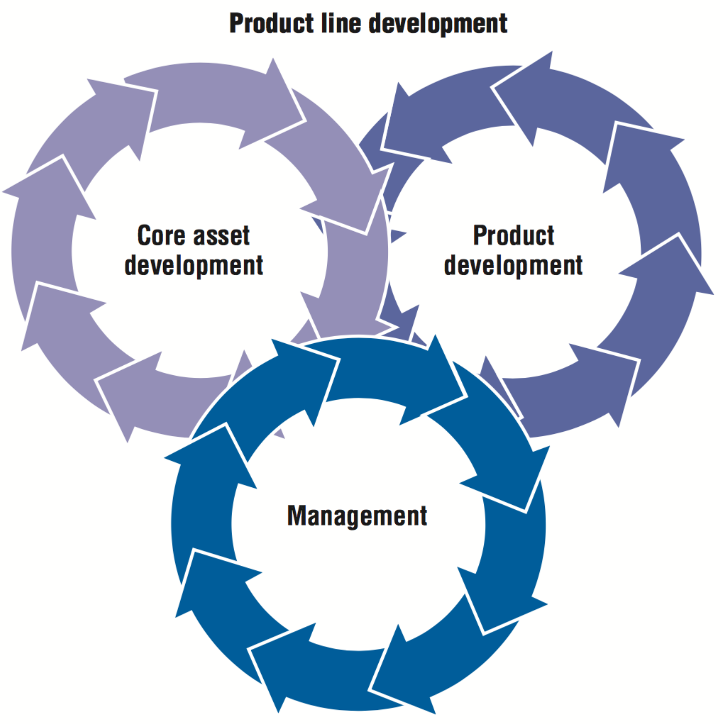


Figure 2.1: Software Product Line Activities [Nor02]

Several works in literature [WL99, PBV05, DSB05, LSR07, ACR09] define a SPLE as a process composed of two sub processes: **domain engineering** or *development for reuse*, and **application engineering** or *development with reuse*. The first consists on defining the core assets while the second exploits the core assets to derive products. Each sub process is divided into a problem space that represents the features describing the products of the SPL and a solution space that implements those features using a variability mechanism. Figure 2.2 shows a framework for SPLE proposed by [PBV05].

2.3.2.1 Domain Engineering

Domain engineering consists on defining the scope of the SPL, the common and variable functionalities provided by the SPL in a reference architecture, a variability model, the reusable artefacts, and the products implemented by the SPL. As per Pohl *et al.* [PBV05], domain engineering is composed of the following sub-processes:

1. **Product management**: determine the scope of the SPL and a product roadmap identifying the common and variable features.

2. **Domain requirements engineering** : collect and document common and variable requirements of the SPL.

3. **Domain design**: define the architecture of the SPL that provides a high-level structure of all the products by means of a variability model.

4. **Domain realization**: design and implement the reusable software components.

5. **Domain testing**: validate the reusable components and verify the implementation of the requirements that they are supposed to implement.

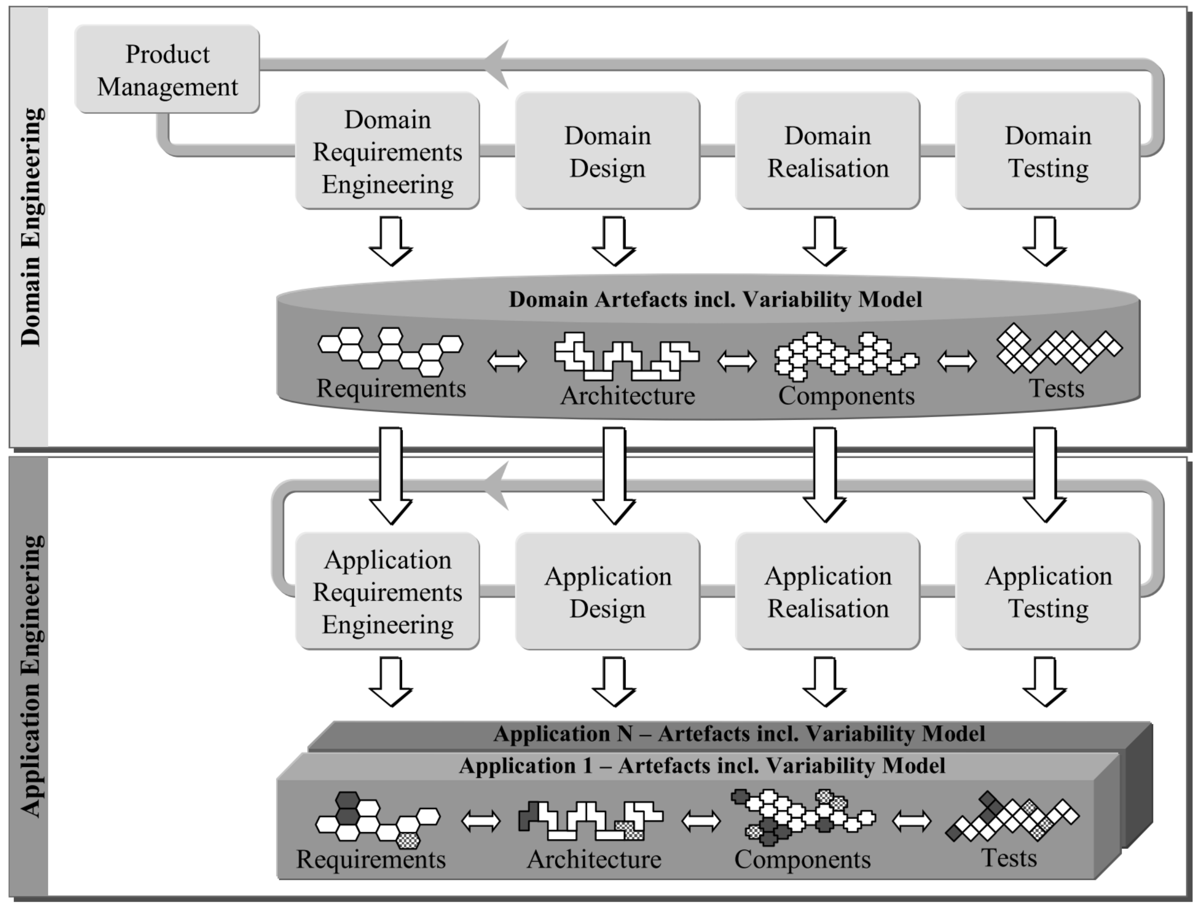


Figure 2.2: Software Product Line Engineering Framework [PBV05]

> **Domain engineering** *is the process of software product line engineering in which the commonality and the variability of the product line are defined and realized.*
>
> [PBV05]

Arango defines domain engineering as a process composed of three domain activites [Ara89]:

1. **Domain analysis**: identify reusable information to be used to specify and implement the system.
2. **Infrastructure specification**: define the infrastructure of the reusable components.
3. **Infrastructure implementation**: implement the reusable components.

2.3.2.2 Application Engineering

Application engineering consists on exploiting the reusability defined in domain engineering in terms of common and variable artifacts to derive products from the SPL.

> “ **Application engineering** *is the process of software product line engineering in which the applications of the product line are built by reusing domain artefacts and exploiting the product line variability.*
>
> [PBV05]

Pohl *et al.* [PBV05] map the domain engineering sub-processes into the following application engineering sub-processes, used to derive an application from the SPL:

1. **Application requirements engineering**: identify the requirements specification of a particular application. A major concern is the identification of particular application requirements that are not part of the domain requirements. This situation may impact the SPL by affecting its artifacts to integrate those requirements and their related artifacts at the SPL domain engineering level.
2. **Application design**: configure the required parts of the domain architecture in order to specify the application design.
3. **Application realization**: create the application by selecting and configuring the reusable software components provided by the domain realization sub-process at the domain engineering level.
4. **Application testing**: validate and verify the derived application against its requirements specification.

2.3.3 Benefits and Drawbacks

As per Bosch [BBS11], the key of success behind SPLs is reusability. He affirms that “the key success factor of software product lines is that it addresses business, architecture, process and organizational aspects of effectively sharing software assets within a portfolio of products” [CBK+13]. Reusing artifacts during SPLE comes up with the following benefits [PBV05]:

— The *reduction of development cost* since an artifact developed once, can be reused in several software products. Although an upfront investment is expensive compared to single system development, an impressive return on investment can be gained after delivering a certain number of systems, as shown in Figure 2.3.

— The *improvement of software quality* since an artifact used in several products is tested and its functionalities are verified in different implementations. Moreover, a detected bug can be corrected and the correction is propagated to the products employing the artifact. Consequently, a high software quality increases the customer trust.

— The *shortening of delivery time* by reducing the development cycle, since many existing artifacts can be reused instead of being developed from scratch. Figure 2.4 shows that developing common artifacts makes delivery of early developed systems slow, but as long as the architecture becomes mature, time to market becomes faster.

— A *better domain understanding* by applying a global analysis of requirements and an architecture covering the whole domain. This can help the organization to train its personnel on using the SPL.

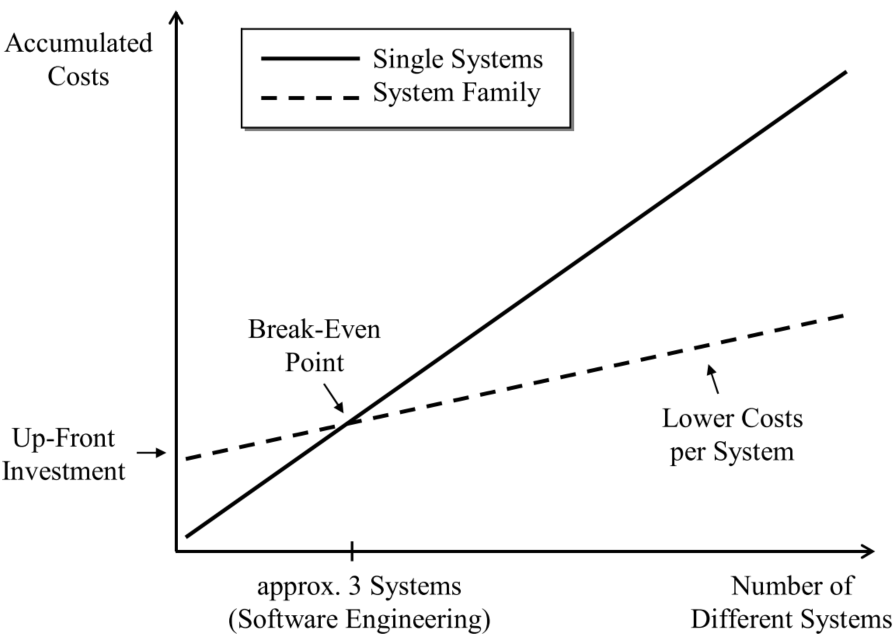


Figure 2.3: Costs for developing n kinds of systems as single systems compared to product line engineering [PBV05]

Despite its long term efficiency, adopting a SPLE approach is a difficult decision to take and can arise the following drawbacks [TH03]:

— A *resistance to change* by the organization engineers and personnel since a shift to a new approach requires mental efforts.

— The *missing of a global view* of the whole architecture of the product line, while it is difficult to find an employee who knows the complete application domain.

— An *expensive initial investment* is required to establish an SPL, since it is specific for a single domain and requires mentality evolution and a high cost in terms of time and budget.

— The *delayed benefits* since an SPL provides a long term return on investment.

— A *difficulty of adoption* of a SPLE approach by small organizations that consider it a risk to take, especially when having short term deadlines with customers. Therefore, SPLE is mainly adopted by large organizations that support a large investment.

— A *difficulty in evolving the SPL* when adopting a classic approach, especially if the evolution is unplanned, because an evolution is about to affect the SPL structure.

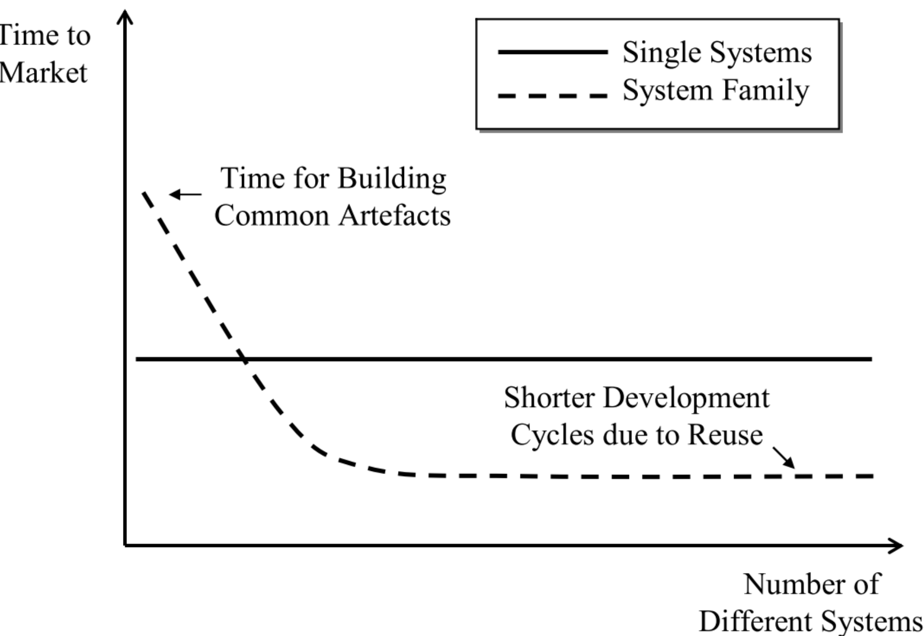


Figure 2.4: Time to market with and without product line engineering [PBV05]

OBJECTIVE 3:
RELYING ON SOFTWARE PRODUCT LINE AS A SUSTAINABLE SOLUTION

As per Weiss and Lai [WL99], SPLE is an up-front investment that performs well for long term, due to the systematic reusability that it provides. Therefore, we consider relying on it as the optimal and sustainable solution when dealing with a large family of products.

2.3.4 Variability Management

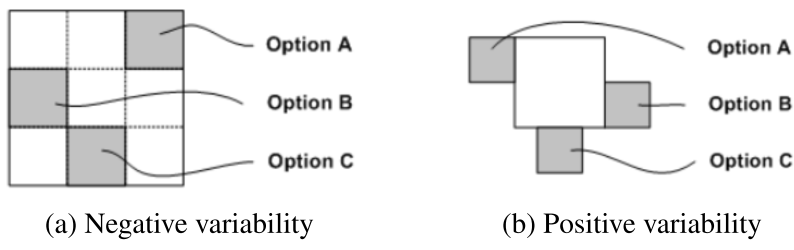
Parnas addressed variability when he defined program families as *"a set of programs is considered to constitute a family, whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the* ***special properties*** *of the individual family members"* [Par76]. The “special properties” stand for variability. Moreover, van Gurp *et al.* defined variability as *"the capability to change and personalize a system"* [vGBS01].

> The **variability** *of a set of software systems or products is the set of differences, described in a structured way, of some or all of their characteristics.*
>
> [AR13]

Voelter and Groher [VG07] identified two kinds of variability: negative and positive.

— Negative variability: unrequired features are deselected from a "maximal" system containing the required features and some additional ones (see Figure 2.5a).

— Positive variability: features are added to a "minimal" core to construct the desired configuration (see Figure 2.5b).



(a) Negative variability (b) Positive variability

Figure 2.5: Negative and positive variability [VG07]

It is important to distinguish between variability at the software level and at the product line level [MP14]. A software variability can be determined by the ability to personalize it or configure it, while product line variability is essential to capture the variable artifacts including requirements, architecture, components and tests, in order to allow their reuse. SPLE requires to define and manage variability during domain engineering, in order to exploit it to derive a variety of products during application engineering. In SPLE, variability is expressed in variation points that are subject to support multiple variable objects called variants [PBV05]. Several variability modeling approaches exist, such as feature models [SHT06], decision models [SRG11] and orthogonal variability model [PBV05].

2.3.5 Feature Model

As per Kang [KCH+90], *a feature is a user-visible aspect or characteristic of the domain*. In SPLE, features are considered as the prime entities of software reuse [BLR+15], providing an abstract representation of the commonalities and variabilities between the SPL products [BSRC10].

> A **feature** *is a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems.*
>
> [HER85]

The term Feature Model (FM) was defined in the Feature-Oriented Domain Analysis (FODA) feasibility study done by Kang *et al.* in 1990 [KCH+90]. A feature model is an abstract presentation of all the SPL products in terms of features. It is defined by a *feature diagram* and a set of *constraints*. A *feature diagram* is an hierarchical tree diagram that shows the features and the relationships between them. A feature diagram has a *root feature*. The relationships between features are defined as *parental relationships* such as *mandatory*, *optional*, *or*, *alternative*, and *cross-tree constraints* such as *requires* and *excludes*. A survey on the different FM semantics is presented in [SHT06] and [SHTB07].

> “ A **feature model** *represents the standard features of a family of systems in the domain and relationships between them.*
>
> [KCH+90]

Figure 2.6a shows a feature model extracted from [BSRC10]. We address this sample mobile phone FM to present and explain the different components, relationships and constraints that may occur in a given FM:

— **Root feature** : the root feature is the feature representing the concept that describes the model and englobes all its characteristics. The feature *Mobile Phone* in the given example englobes the characteristics *Calls*, *GPS* and so on.

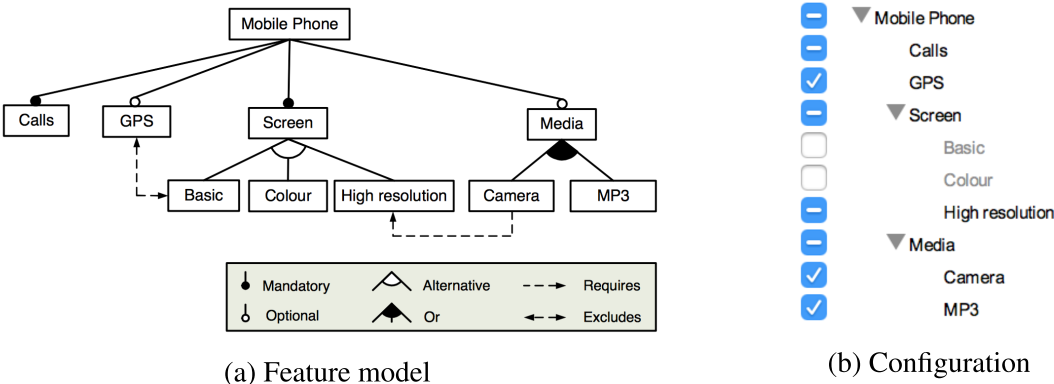
— **Parental relationships** :

- **Mandatory**: a *mandatory* relationship between a parent feature and a child feature means that the child feature is present in all products where the parent feature is present. The given FM example shows that all products — which are mobile phones — allow *Calls* functionality and have a *Screen*.
- **Optional**: an *optional* relationship between a parent feature and a child feature means that the child feature can optionally be present in all products where the parent feature is present. Some, but not all mobile phones in the given example support *GPS*.
- **Alternative**: an *alternative* relationship between a group of features and a parent feature means that only one feature of the group can be selected when the parent feature is present. A mobile phone can have one and only one screen that can be either *Basic*, *Colour* or *High resolution* screen.
- **Or**: an *or* relationship between a group of features and a parent feature means that one or more features of the group can be selected when the parent feature is present. A mobile phone that supports *Media* can have *Camera*, *MP3* or both.

— **Cross-tree constraints**:

- **Requires**: a *requires* relationship directed from a feature *f1* to a feature *f2* means that each product containing *f1* must contain *f2*. A mobile phone that includes a *Camera* must have also a *High resolution* screen.

– **Excludes**: an *excludes* relationship between two features *f1* and *f2* means that *f1* and *f2* cannot be present in a same product. No mobile phone has *GPS* with a *Basic* screen.



(a) Feature model (b) Configuration

Figure 2.6: A sample feature model and a configuration [BSRC10]

OBJECTIVE 4:
MODELING VARIABILITY THROUGH A FEATURE MODEL

In our dissertation, we are interested in constructing a feature model, allowing the configuration of existing products by selecting the required features. Moreover, we are interested in exploring negative and positive variability in order to enable the construction of new products.

2.3.6 Product Configuration

A configuration is a selection of a collection of features that respect the constraints imposed by the FM and generally reflect an SPL product. It is defined by Junker as *"the task of composing a customized system out of generic components"* [Jun08].

> “ A final fully-specific feature model with no points for further customization is called a **configuration** *of the feature model based on the selected features.*
>
> [BEG+11]

A configuration process is done during application engineering. As per Faltings et Freuder, software developers use standardized software components that can be configured into products to respond to the customer requirements. They consider that a configuration must be correct, optimal and quickly produced in order to maintain customers [FF98].

Li *et al.* [LYSL07] define the product configuration as a design activity that takes the configuration models and the requirements as an input and outputs a configuration result oriented to the ultimate product.

According to Botterweck and Pleuss, *"a product is defined by a product configuration, which resolves the variability by selecting from the given variants while considering the defined constraints. In the case of a feature model, this is done by selecting or eliminating features"* [BP14].

Figure 2.6b shows a configuration of the corresponding FM in Figure 2.6a. Selecting the feature *GPS* enforces an automatic deselection — if selected — of the feature *Basic* and disables its selection, since an *exclude* constraint exists between *GPS* and *Basic*. Selecting the feature *Camera* enforces the selection of the feature *High resolution* and deselects — if any is selected — and disables the features *Basic* and *Colour*, since a *requires* constraint exists between *Camera* and *High resolution*. We call this configuration a *valid configuration*, since it reflects the implementation of the selected features in one of the SPL products.

As per Czarnecki *et al.* *"a feature model describes the configuration space of a system family"* [CHE04]. A *configuration space* is defined as the set of all possible configurations from a FM [Mar16]. Table 2.1 shows the configuration space corresponding to the FM of Figure 2.6.

Table 2.1: A configuration space corresponding to the FM of Figure 2.6

Configuration	Mobile Phone	Calls	GPS	Screen	Basic	Colour	High resolution	Media	Camera	MP3
1	✓	✓		✓	✓					
2	✓	✓		✓	✓			✓		✓
3	✓	✓		✓		✓				
4	✓	✓	✓	✓		✓				
5	✓	✓		✓		✓		✓		✓
6	✓	✓	✓	✓		✓		✓		✓
7	✓	✓		✓			✓			
8	✓	✓	✓	✓			✓			
9	✓	✓		✓			✓	✓	✓	
10	✓	✓	✓	✓			✓	✓	✓	
11	✓	✓		✓			✓	✓		✓
12	✓	✓	✓	✓			✓	✓		✓
13	✓	✓		✓			✓	✓	✓	✓
14	✓	✓	✓	✓			✓	✓	✓	✓

2.3.7 Product Derivation

Features are an abstraction of the implementation of reusable assets [CA05]. A product derivation is based on the product configuration and the feature mappings to the corresponding assets [BP14].

> “ **Product derivation** *is a key process in application engineering and addresses the selection and customization of assets from the product line (utilizing the provided variability) to satisfy customer or market requirements.*
>
> [DSB05]

Product derivation can lead to an automated derivation of an existing product, in case the set of required features correspond to a valid configuration. However, to respond to the arising customer requirements, negative and positive variability can be used to derive new products [VG07] (see Figure 2.5). The former is achieved by removing some unrequired features from a maximal product, that implements the required features and more. While, the latter is achieved by using a minimal core to add additional features required for the derivation and not contained in the core [Mar16].

2.4 Summary and Contribution Decisions

In this chapter, we have seen that reuse is the solution to respond to the mass customisation necessity. Therefore, any contribution is supposed to explore reuse as much as possible to gain its efficiency. Earlier in this chapter, we presented two approaches for reuse which are C&O and SPLE, and we highlighted on the benefits and drawbacks of each. Before making our contribution choices, we compare the presented approaches according to literature, regarding the development **challenges** identified at the early beginning of the chapter:

— **Cost & Effort:** it is most likely to face a high development cost and effort in the early development phase in both C&O and SPLE, since the artifacts are usually created from scratch. According to Dubinsky *et al.*, when C&O is adopted, the cost and effort decrease due to the quick and easy clone process, however, when the number of products becomes significant, the cost and effort to derive a new PV increase dramatically, since it becomes very difficult to determine which PV to use as the source of clone [DRB+13]. Moreover, the required functionalities for the product to derive might be propagated into different PVs which can make the collection of the required functionalities a tedious task [LBC16]. For SPLE, according to Pohl *et al.*, the development of artifacts in domain engineering phase makes the cost and effort expensive for the starting phase, however, as long as the SPL becomes mature, the cost and effort decrease due to reusability [PBV05].

— **Quality:** in C&O, software quality starts high for early delivered PVs, since clones are subject to test and corrections. However, the quality starts to decrease when a considerable number of variants is produced, especially when organizations are committed into short delivery periods. Further, quality decreases since a correction done on a variant might not be propagated to other variants, due to the missing traceability of the corrected functionality in the other variants [FLLHE14]. On the other hand, software quality in SPLE is considered higher than C&O, since artifacts are subject to test during domain engineering and later on during application engineering before delivering the product. This quality keeps increasing as long as the SPL becomes mature since artifacts reused in several products gain more trust to PVs [PBV05].

— **Delivery:** in SPLE the delivery time for the early developed products is considered very high compared with C&O due to the up-front investment of creating artifacts that can be reused by several products. The delivery time decreases in C&O due to cloning, then increases dramatically when identification of the variant to clone becomes ambiguous and the required functionalities propagated on different variants without being able to easily identifying those variants. This makes the collection process of those functionality from the existing variants a very tedious mission that delays considerably the delivery time [DRB+13]. In SPLE, reusability of artifacts allows to maintain a low delivery time [TH03].

— **Maintenance:** the maintenance effort in C&O is proportional to the number of variants produced. As long as the number of variants increases maintenance becomes difficult [FLLHE14]. From a side, its difficult to identify the variants to propagate the maintenance in, and from other side it is expensive to repeat the same maintenance process on the identified variants that require the modification. For SPLE, it is considered much less expensive to make a maintenance since architecture of the SPL is well structured and reusable components are identified [PBV05]. However, it is still challenging to propagate a maintenance on several products, since a functionality may behave differently in different products and therefore the maintenance must be done precisely and supplemented by testing [BP14].

The comparison between C&O and SPLE shows that C&O performs well in early development stages, but turns into an expensive solution when the family of products delivered become rich. On the other side, SPLs are adopted due to their long term return on investment and they are considerably expensive in the early development stage.

Objective 5:
Combining clone-and-own and software product line

Since, from a side C&O is the optimal approach for starting the development of a family of software products, and SPLE is a sustainable solution, an effective contribution must adopt C&O in early development stages then integrate the family of products into a systematic SPL when it starts to consolidate its foundation. Nevertheless, C&O remains essential when unplanned new products have to be derived.

Minimizing cost and effort can be achieved by guiding the identification of the possible sources of clone and the selection of the required assets for derivation. In fact, when identifying the assets that do not require modification, we save the derivation from lacking into an intensive cloning process. Relying on the SPL artifacts as source for cloning improves quality and trust in code, since it corresponds to code tested and subject to reuse. SPLs enable an automated derivation of existing products ensuring fast delivery. Moreover, employing C&O to derive a new product from the SPL can shorten the delivery time if the necessary guidance is offered. Maintenance within an SPL context is favorable since artifacts are used in several products, hence, integrating the newly derived products in the SPL is a must.

☞ In **Chapter 4** we demonstrate the migration process of our approach, in which we propose the migration of PVs into an SPL.

CHAPTER 3

RELATED WORK

Contents

3.1 Software Product Line Adoption **32**
3.1.1 Migration Approach 32
3.1.2 Migration Steps 32
3.1.2.1 Identifying Features 34
3.1.2.2 Mapping Features to Assets 36
3.1.2.3 Constructing a Feature Model 40
3.1.3 Migration Moment 40
3.2 Product Derivation Support **41**
3.3 Software Product Line Evolution **45**
3.4 Summary and Contribution Choices **48**

3.1 Software Product Line Adoption

Migrating from the development of single and independent software product variants using clone-and-own approach, to a structured software product line raises the following questions:

1. *What is the approach to adopt for migration?*
2. *What are the required steps to accomplish the migration?*
3. *What is the accurate moment to proceed the migration?*

3.1.1 Migration Approach

Krueger [Kru01] distinguished three models for adopting software mass customization by means of a software product line, which are *proactive*, *reactive* and *extractive*.

— ***Proactive approach:*** the organization builds a complete SPL that takes into consideration the current and expected customer requirements that belong to the scope of the SPL. The common and variable features are identified, their corresponding assets are implemented, and the possible products are defined. This approach requires the maximum effort from the organization, because the SPL is fully created before starting the product derivation process.

— ***Reactive approach:*** the organization creates a minimal SPL to start deriving products. The SPL is incrementally evolved in reaction to the upcoming customer requirements. The reactive approach imposes less initial effort compared with the proactive approach.

— ***Extractive approach:*** the organization uses existing product variants in order to extract the common and variant artifacts, identify the features and create a corresponding SPL. This approach allows a quick SPL adoption by the high level of software reuse that it provides.

OBJECTIVE 6:
EXTRACTIVE MIGRATION APPROACH

In our approach, we are interested in migrating existing PVs developed in C&O into an SPL. Thus, we care about performing an extractive migration approach.

3.1.2 Migration Steps

Anwikar *et al.* [ANCM12] define the migration as a process of three phases:

1. **Detection phase:** extract from source code information about the structure and the various functionalities implemented.
2. **Analysis phase:** use the information gathered in detection phase to create new partitions that identify and separate the features.

3. **Transformation phase:** use the knowledge gathered in analysis phase to produce the SPL by transforming the legacy applications into separated layered modules of source code to allow artifacts reuse.

Martinez *et al.* [Mar16] identify the activities required for achieving an extractive SPL adoption as follows (see Figure 3.1):

1. **Feature identification and naming:** Martinez considers the identification of the features implemented in the existing product variants as part of the initial activity. Despite of being possible that the software developer manually identifies the features by traversing the artifacts of the product variants, Martinez highlights on the need for a mechanism to identify those features when dealing with complex products, where functionalities are provided by different stackholders. A manual identification of the features is supplemented by a manual naming of the identified features. However, when an identification mechanism is performed, Martinez *et al.* offer a word cloud visualization mechanism to help software developers in naming the identified features.

2. **Feature location:** if features are manually identified by software developers, feature location has to be performed. This activity consists on the identification of the assets associated to each feature.

3. **Feature constraints discovery:** this activity consists on discovering the constraints between the identified features, in order to ensure the validity of the configurations of the FM to be created.

4. **Feature model synthesis:** after identifying the features and the constraints between them, a structured FM is created, allowing the automated derivation of existing products and the ones reflecting valid configurations made on the generated FM.

5. **Reusable assets construction:** the implementation elements associated to each feature are used to construct the reusable assets in order to be employed during future derivations.

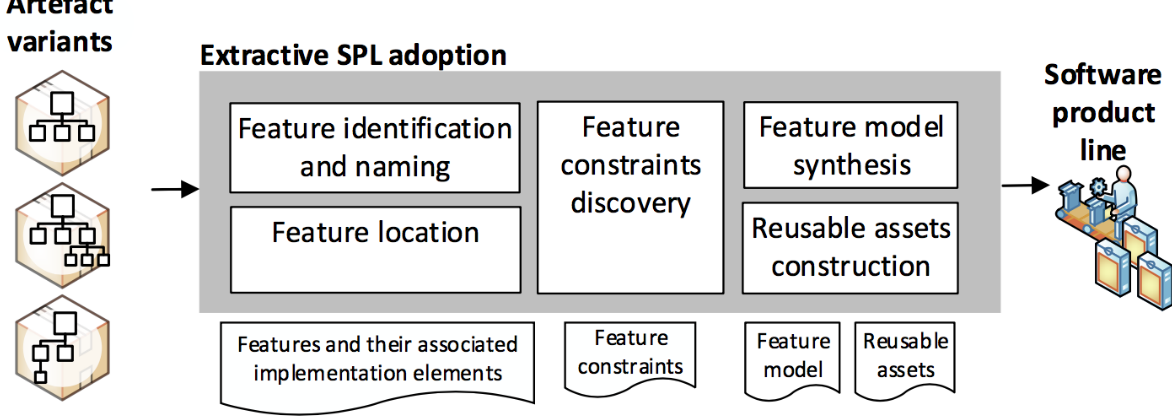


Figure 3.1: Relevant activities during extractive SPL adoption for leveraging artefact variants [Mar16]

Al-Msie'Deen *et al.* [AM14] defined SPL adoption as a process of reverse engineering a feature model from software variants. Their approach consists of the following steps:

1. Extracting feature implementations from source code and use-case diagrams of the software variants.

2. Documenting the mined feature implementations using use-case diagrams.

3. Constructing the feature model and its constraints from the mined and documented features.

Based on the related work approaches, we synthesize three main steps to be performed in order to construct the SPL during the migration process:

1. Identifying features

2. Mapping features to assets

3. Constructing a feature model

In the upcoming subsections, we detail the different techniques adopted in literature to perform each step, and we identify our contribution choices regarding each step.

3.1.2.1 Identifying Features

Martinez *et al.* [Mar16] consider that it is very idealistic to assume that the features implemented by the PVs are identified or can be obtained from domain knowledge prior to migration. When features are not manually identified, a technique called *feature identification* is performed in order to identify the features implemented by the PVs and their corresponding implementation elements [AG06].

> “ **Feature identification** *is defined as the activity of identifying the source code constructs implementing a given feature.*
>
> [AG06]

Martinez *et al.* developed a framework for extractive SPL adoption called *BUT4Reuse* that stands for *Bottom-Up Technologies for Reuse* [MZB+17, Mar16, MZB+16, MZB+15b, MZB+15a, MZKLT14, MZM+14]. The framework achieves the process presented in Figure 3.1. *BUT4Reuse* performs feature identification if features are not identified prior to migration. The process analyzes the product variants artifacts and specifies separated blocks that stand for the identified features. The framework illustrates the common and varying blocks. It constructs the blocks separately, to be known as reusable assets and allows to associate each of them to a feature. *BUT4Reuse* helps domain experts in feature naming task, through the word clouds visualization functionality that it provides. The content of the identified blocks is analyzed, and a weight is given to each word corresponding to its occurrence in the block content. As represented in Figure 3.2, the word having the highest occurrence will be displayed with the largest font size, and so on.

When feature identification is performed, feature naming is required in order to assign a name for each identified feature from the analyzed artefacts. While some approaches propose

automatic feature naming [DDH+13, IRBW16], or help the domain experts in choosing the corresponding name [Mar16], other approaches [YPZ09, AM14, MAGDC+16] keep feature naming as a manual task. This task might be expensive, since it requires a lookup on the artifacts corresponding to each identified feature, in order to designate a meaningful naming to the features. Kästner *et al.* [KDO14] developed a tool called *variability mining*. The goal behind the tool is to facilitate the adoption of an extractive SPL by helping the developer to locate, document and extract the implementations of the features in a semi-automatic way. Liu *et al.* [LBL06] developed a theory based on feature oriented refactoring, which consists on decomposing a program into features. The theory identifies the relationship between features and their implementations. Approaches proposed in [LBL06, KDO14] identify feature information from a single product in contrary to [Mar16] that concentrate on identifying features from several product variants.

Figure 3.2: Word clouds visualization functionality for feature naming [Mar16]

Several works in literature that propose to migrate PVs into an SPL [AM14, RC12], or to support C&O [FLLHE15, RC13a, LBC16], assume that features are determined by domain experts [FLLHE15] or extracted from requirements documentation [LBC16] or use-case diagrams [AM14].

OBJECTIVE 7:
BUSINESS-LEVEL FEATURE IDENTIFICATION

Feature identification techniques rely on implementation artifacts without taking into account business-level features. Moreover, the proposed feature naming techniques [Mar16, DDH+13, IRBW16] rely on source code and implementation artifacts to extract or propose names for the identified features, which might reflect a technical not a business-level perception of the implemented features. Kang *et al.* define features as *"the attributes of a system that directly affect end-users"* [KCH+90]. In our dissertation, we do not contribute in or employ feature identification, since we consider that features reflect the user business requirements, and therefore, they are supposed to be given by domain experts instead of being identified through automatic feature identification techniques.

☞ In the migration process that we propose in **Chapter 4**, we assume that features implemented by each product are known prior to its migration into the SPL.

3.1.2.2 Mapping Features to Assets

Ji *et al.* affirm that "to effectively evolve and reuse features, their location in software assets has to be known" [JBAC15]. Hence, when features are identified, the next step is to locate the implementation of the features in the variant artifacts. In other words, it is about mapping the identified features to the PVs assets.

Several works in literature proposed to analyze and compare the artifacts of the product variants to identify their common and variable parts [Mar16, FLLHE14, AmSH+13, RC12]. Thus, determining the coexistence between the features and the identified parts of the product variants allows to establish the mapping between them. The identified parts are called *blocks* in [ANCM12, Mar16], *clusters* in [YPZ09], *regions* or *parts* in [RC12], *modules* in [FLLHE14, MAGDC+16], and *atomic blocks* or *object-oriented building elements sets* in [AM14].

> “ A **block** *is a set of implementation elements of the artefact variants that are relevant for the targeted mining task.*
>
> [MAR16]

Feature location is an activity that identifies mappings between features and assets. Dit *et al.* define feature location as "the activity of identifying an initial location in the source code that implements functionality in a software system" [DRGP13].

> “ **Feature location** *techniques aim at locating software artifacts that implement a specific program functionality, a.k.a. a feature.*
>
> [RC13B]

Surveys on feature location techniques were conducted by Rubin and Chechik [RC13b], Dit *et al.* [DRGP13] and Assunção *et al.* [AV14] representing and comparing the different techniques available in the domain.

Rubin and Chechik [RC13b] made an overview of twenty-four feature location techniques and their underlying technologies such as Formal Context Analysis (FCA) [Bel08] and Latent Semantic Indexing (LSI) [DDF+90]. They show that all techniques studied and compared in their survey treat the products as individual independent entities and not as a family of related

entities. Hence, they affirm that taking into consideration commonalities and variations of the related products can improve the accuracy of the adopted techniques by initially partitioning the code into unique part to the product and shared parts. Moreover, they highlight on the need of an incremental analysis of the traceability between features and asset to save the efforts of re-analyzing.

The survey made by Assunção *et al.* [AV14] shows that the research about feature location techniques is on a continuous growth since 2010. The survey shows that 47% of the related techniques focus on detection phase, 43% on analysis and 10% on transformation. Most techniques consider source code artifacts as the main input, with some focus on other artifact types such as design models, requirements and documentation. The output of the proposed techniques can be divided into commonalities and variability information, feature mapping to elements, feature models, and source code reorganization.

OBJECTIVE 8:
GLOBAL MAPPING BETWEEN FEATURES AND ASSETS

Based on the conclusion made by Rubin *et al.* [RC13b] in their survey about feature location, our approach aims to establish mappings between features and assets, by taking into consideration the entire product line instead of treating products separately to identify the mappings.

Beside providing feature identification, Martinez *et al.* enable in their approach to perform directly feature location in case features are known [MZKLT14, MZM+14, MZB+15b, MZB+15a, Mar16, MZB+16, MZB+17]. When performing feature location, their framework *BUT4Reuse* works on several artifact type variants such as Java projects, UML, JSON, text files, images and more. For each artifact type an adapter is required and the framework supports currently fifteen adapters. It takes as an input the existing product variants and chooses automatically the appropriate adapter corresponding for the artifact type of the variants.

AL-Msie'Deen *et al.* [AM14, RSH+13, SHU+13] proposed an approach called *REVPLINE* that stands for *REengineering Software Product Variants into Software Product LINE*. *REVPLINE* aims to mine features from object-oriented based product variants. The approach uses Formal Concept Analysis to identify the common blocks and blocks of variations between the product variants and relies on Latent Semantic Indexing [DDF+90] to determine the similarity between the object-oriented building elements. The approach exploits the source code of the mined features and use cases in order to provide a name and a description to the feature. This information is used then to construct the feature model of the SPL.

Fischer *et al.* [FLLHE14], in the context of their approach to enhance C&O, consider that features are provided by domain experts with the product variants in which they are implemented. They define an algorithm to compare the provided PVs and identify their commonalities and variability. The identified parts called *modules* are categorized into *base modules* that refer to artifacts implementing a feature without any feature interactions, and *derivation modules* that refer to artifacts implementing feature interactions. Feature interaction occurs when two or more features are totally or partially implemented by a common artifact. Similarly to the approach

proposed by Martinez *et al.* [Mar16], this approach supports also different artifacts types, and employs the appropriate adapter corresponding for the artifact type of the variants.

> “ *A* **feature interaction** *is a situation in which two or more features exhibit unexpected behavior that does not occur when the features are used in isolation.*
>
> [AK09]

A language-independent approach called *ExtractorPL* is proposed by Ziadi *et al.* [ZHP+14], as a reverse engineering approach for creating an SPL from existing software variants. The approach consists on extracting the features by identifying the variability among the product variants, regrouping the identified features in a feature model, and mapping the features to their corresponding source code blocks.

OBJECTIVE 9:
LANGUAGE-INDEPENDENT MAPPINGS IDENTIFICATION

The specific artifact type adapters employed in [MZB+15b] and [FLLHE14] perform well when the family of products consists of monoglot variants or specific adapters exist for their corresponding artifacts. In our approach, we give interest as well in polyglot variants such as web applications, that are written using different languages and consist of artifacts of different formats, where some may contain blocks of code of different languages. Therefore, we aim to introduce a language-independent mechanism to establish mappings between features and assets when polyglot variants are used, without limiting our approach to the proposed mechanism, especially when monoglot variants are integrated and specific adapters are pertinent to capture their mappings.

According to Botterweck and Pleuss [BP14], mapping between features and the assets implementing them is required. This mapping is more complex than a one-to-one relationship, since a feature is most likely mapped to different assets. Further, the theory proposed in [LBL06] identifies the relationship between features and their implementations, and proves that a feature can have different implementations.

Objective 10:
Capturing features interactions

We consider that a feature can be implemented through several assets, while an asset can contribute in the implementation of several features. Therefore, a many-to-many relationship between features and assets is likely to occur. In our dissertation, we give interest to deal with similar cases. Moreover, we consider that a feature can have different implementations, due to the interaction between the feature and the other features implemented in the corresponding product. The presence of a feature in a product, and its absence in another, might affect the implementation of other features implemented by the same products due to features interactions. One of the strength points of the approaches proposed in [MZB+15b] and [FLLHE14] is that they take into consideration features interactions. Hence, in our approach, we are interested in capturing features interactions.

Rubin *et al.* proposed a *merge-refactoring* framework for managing cloned product variants [RC12, RKBC12, RC13a, RCC13]. They highlighted on the necessity of representing the implemented functionalities of the product variants as features and they identified a set of operators to manage and maintain such products either by sharing features between them or by representing them in an SPL.

Narwane *et al.* [KNGDNK+16] define an SPL as as triple composed of *specifications* where features are represented in a feature model, *implementations* where assets are represented in a components model and *traceability* relationships between features and assets. They define reasoning operators to identify the state of the SPL and investigate traceability between features and assets. For instance, an SPL is *valid* if there exists a specification and an implementation. In another example, an implementation *realizes* a specification, if the implementation assets implement all and only the features of the specification. Moreover, an implementation *covers* a specification, if the implementation assets implement all and more than the features of the specification.

In our approach, we are interested in integrating the functionalities of some operators from [RC13b] and [KNGDNK+16], in a context that serves the objectives of our contributions. For instance, operators such as *findFD* from [RC13b] to find the features implemented by a variant, *realizes* and *covers* from [KNGDNK+16] are necessary to fulfill the objectives of our approach.

Ji *et al.* [JBAC15] propose a lightweight code annotation approach. Their approach consists of embedding annotations about the features directly into the source code of the assets. A study of their approach on a product line of cloned product variants, shows that the cost of adding and maintaining annotations is small compared to the actual development cost. Hence, their approach provides a low-cost feature location through annotations, and reduces maintenance efforts, since annotations co-evolve with assets.

We recognize that the approach proposed by Ji *et al.* [JBAC15] is a lightweight and effective solution that maintains incremental mappings between features and assets. However, we consider that to gain the effectiveness from their approach, it must be adopted from scratch, from the

moment where the assets of the first product are developed. However, in our approach, we are interested in migrating pre-developed product variants into an SPL and thus, annotating source code of several variants that share features in common can be considered tedious and error-prone. Otherwise, the proposed annotations approach can give an effective return on investment if done from scratch.

☞ In **Chapter 4** we define a language-independent mechanism to identify mappings between features and assets, and we call them *correlations*.

3.1.2.3 Constructing a Feature Model

A method to extract a domain feature model from existing product variants is proposed by Yang *et al.* [YPZ09]. The method uses data access semantics and Formal Concept Analysis (FCA) [Wil96, Bel08] to establish the relationship between the features and their implementations. Yang *et al.* consider feature naming as a manual task to be done by domain experts by examining the generated features and evaluate its business meaning.

Martinez *et al.* [Mar16] proposed an approach for constraints discovery that can be generalized on several artifact types. In their approach, constraints between the identified features are automatically discovered in order to allow the creation of a feature model that enables valid configurations. The feature model generated by the proposed framework *BUT4Reuse*, is explored using the *FeatureIDE* tool [TKB+14] in order to configure new products.

Acher *et al.* [Ach11] designed a set of composition and decomposition operators to manage multiple FMs. For instance, a *merge* operation permits to construct a feature model by merging several ones [ACLF13]. Moreover, they developed a textual language called *FAMILIAR* with a tool offering a practical solution to manage FMs.

☞ In **Chapter 4** we construct the SPL feature model by applying a *FAMILIAR merge* operation [ACLF13] on the feature models provided with the migrated PVs. Similarly, as presented in **Chapter 7** after the integration of a newly derived product, this merge operation is used to merge the product FM with the SPL FM, to keep the latter up-to-date.

3.1.3 Migration Moment

As presented in **Chapter 2**, the large investment in terms of cost and time for building an SPL from scratch, made it an undesired option for an initial development approach. Consequently, an expensive migration to SPL adoption would be undesired too. To determine the accurate moment to migrate PVs into an SPL, we refer back to Figure 2.3 that compares the development cost between single systems and system family. Figure 2.3 shows that approximately after the development of three software systems, the product line engineering approach can provide a lower cost per system. We refer to this indicator to set the following hypothesis:

OBJECTIVE 11:
SMOOTH MIGRATION IN THE ACCURATE MOMENT

Migrating PVs into an SPL is preferred to be done once the family of products starts to consolidate its foundation. We mean by this, when the family of products is composed of few products, implementing a set of functionalities, where some of them are common and others are variable. We do not set a specific number of products to determine the accurate migration moment, but we consider that once the number of products reaches three, it is recommended to migrate them into an SPL, in order to overtake the challenges confronted during C&O. A smooth and quick migration process is required, in order not to affect the productivity of the development unit.

3.2 Product Derivation Support

BUT4Reuse [MZB+15b, MZB+16], the framework proposed by Martinez *et al.* generates a feature model that allows the automated derivation of the product variants migrated into an SPL. In addition, the generated FM permits an automated derivation of new variants as long as the set of selected features during the configuration of the FM does not break the FM constraints. The approach does not guarantee the derivation of a complete product, and does not provide a support to complete it.

Fischer *et al.* developed the *ECCO* approach, which stands for *Extraction and Composition for Clone-and-Own* [FLLHE14, FLLHE15, LHLE15, LLHE16]. They give interest to developers that initially adopt C&O to produce software variants. According to them, management and maintenance of the software variants soon becomes ineffective [FLLHE14]. Therefore, they propose *ECCO*, as an approach that integrates software variants to provide a systematic reuse and helps software engineers to derive new variants by proposing the necessary hints. The proposed hints can either inform developers of the existence of surplus artifacts that have to be removed manually, or the need of reordering some artifacts in case several possible ordered were detected. The workflow of the *ECCO* approach is shown in Figure 3.4. First, it takes as an input the existing product variants, and the features assigned to each variant. Therefore, feature identification is considered as a manual task and the initial task performed by *ECCO* is feature location. This task consists on the extraction of the implementation artifacts called modules and tracing them to their corresponding features and features interactions. The traced information is used to identify the relationships between the modules called associations. A configuration of features leads to the composition which uses the associations to construct a product. If the constructed product refers to an existing variant, it is automatically generated as a complete product. Otherwise, it is considered as a composed incomplete product and requires a manual completion. A product is considered incomplete, either if some features or feature interactions did not exist earlier in existing variants, or they always appeared together in same variants, and need to be separated. During this phase, *ECCO* provides the software engineer with some guiding hints to finalize the product construction. Such hints can inform the software engineer for example, that some modules had never appear separately and need to be separated, or for a set of artifacts multiple ordering options are available. When a new variant is completed,

it is integrated in *ECCO*, where it can be used in future product derivations.

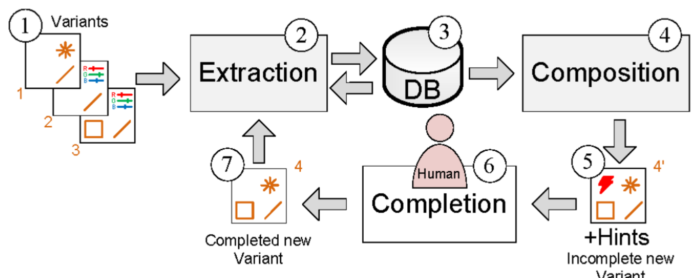


Figure 3.3: The *ECCO* workflow [FLLHE15]

Linsbauer *et al.* [LELH16, Lin16] transformed the *ECCO* approach into a feature-oriented and distributed version control system (VCS). The main operations provided are *commit* and *checkout*. The *commit* operation allows to integrate a product variant into the VCS. The product variant is provided with the features that it implements. The project *repository* contains the different artifacts extracted from the product variants. The *checkout* operation is used to derive an existing variant and possibly a new variant. A new variant is derived by selecting a set of features that are not implemented in a single variant. In this case, the VCS provides the software engineer with some hints to guide him construct the new variant. Moreover, the graphical user interface of the *ECCO VCS* supports the derivation of new variants by artifacts selection, and not only by features selection. This functionality gives the software engineer more possible configurations. The workflow of the *ECCO VCS* is shown in Figure 3.4.

Despite the efficiency of the approach proposed by Fischer *et al.* [FLLHE14] in enhancing and supporting the derivation of PVs based on C&O, we consider that it limits the freedom that software engineers are supposed to benefit from when adopting C&O. Their approach seeks to provide a systematic reuse by automating the derivation of product variants whenever possible. Although they allow developers to make their decisions when completing an incomplete product based on the proposed hints, they offer them a single derivation solution which is the constructed product. In other words, their approach limits developers' decisions to low level decisions, while the proposed solution is imposed on them. When the tool proposes an uncomplete product, that requires a manual completion, the software developer accomplishing this task might find in hands some proposed assets that she is not familiar with or never worked on. Hence, developers are not able to recognize how the proposed product was constructed, and to which product variants the assets collected to construct it belong. In fact, developers lose decision-making and freedom to build a new unplanned product by themselves, and thus, they lose somehow the ownership meaning of the derived product.

Lapeña *et al.* [LBC16, LnFPC16, BLnC16, LnFPC17] consider that C&O consumes high amounts of time and effort. Therefore, they propose an approach called *Computer Assisted Clone And Own (CACAO)* to assist C&O. This approach takes as input the product variants in addition to their documented natural language requirements. The goal of the approach is to support software engineers when deriving new product variants. Hence, when a new product has to be derived, the requested documented requirements are provided. *CACAO* extracts keywords from the new requirements and from the existing product variants requirements using POS Tagging techniques [Hul03]. Next, it detects which existing product variants are closer to the product

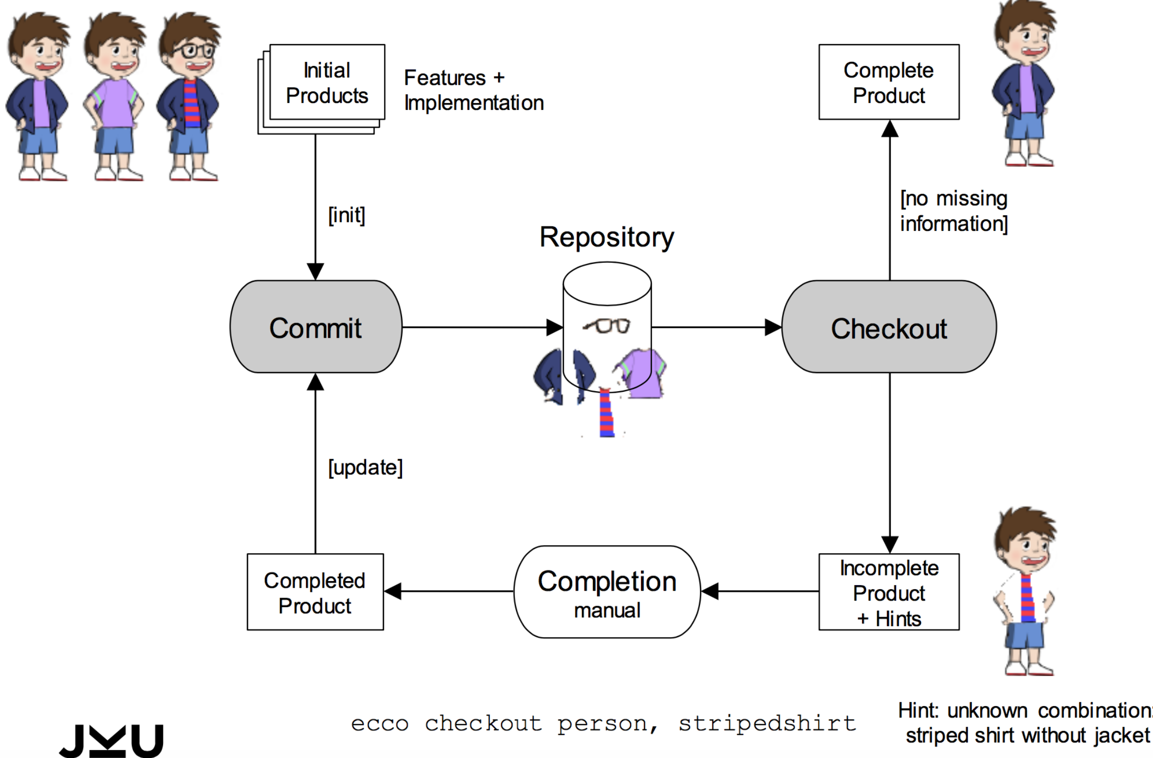


Figure 3.4: The *ECCO* version control system workflow [Lin16]

variant to derive, in terms of requirements by employing Coarse Grain Latent Semantic Indexing (CG-LSI) [DDF+90, LFL98]. Finally, it determines which are the source code methods of the existing product variants, that are closer to the requested requirements, by employing Fine Grain Latent Semantic Indexing (FG-LSI). Hence, *CACAO* provides ranking at two levels, which are products level and methods level. The product relevancy ranking allows software developers when deriving a new product, to decide if they rely on the product variants that they are familiar with, a mixture between known and unknown variants, or non-familiar products having the highest ranking. The code relevancy ranking provides software engineers with the most relevant methods for each of the new product requirements.

We express our interest in three main contributions of the approach proposed by Lapeña *et al*. The support provided to software developers to help them in choosing the relevant products and methods to derive a product, by first proposing the different possible solutions, second keeping the decision-making to software developers without imposing a solution on them, and third by ranking the products and methods to help them make their decisions. On the other hand, we consider that their approach misses an additional details level, that can be achieved by providing the several combinations of products that can lead to the new product to derive, with the operations to be performed on the identified methods such as removing or extracting features from them.

OBJECTIVE 12:
SUPPORTING DERIVATION WITH THE POSSIBLE SCENARIOS

In our approach, we aim to guide software engineers in deriving new products, by providing them with the different possible scenarios that they can rely on to derive the new product, and the different possible operations to perform on their assets such as removing or extracting features from them.

☞ In **Chapter 5** we demonstrate how we support the derivation of new product variants by providing for each configuration its configuration scenarios and operations to perform.

OBJECTIVE 13:
COST-ESTIMATED DERIVATION

Based on the required features, many might be the scenarios that can be achieved to derive a new product. In order to tackle complexity and facilitate the choice of software engineers in selecting the relevant products and operations to perform to derive the desired product, we aim to define indicators that allow to estimate the cost of the proposed operations, thus, software engineers can rely on the estimated cost of the operations to choose the appropriate ones to construct the new product.

☞ In **Chapter 6** we define indicators that allow to estimate the cost of the operations to perform. We provide cost-estimation to software engineers to facilitate their choice.

OBJECTIVE 14:
SOFTWARE ENGINEERS AS DECISION MAKERS

We consider that automated derivation can degrade ownership level and trust of developers in the newly derived products. To maintain the freedom that software developers benefit from when deriving new products based on clone-and-own, we provide them with the possible solutions and keep to them the decision-making in choosing the relevant solution. Since the number of possible solutions might be large, we aim to tackle complexity and facilitate decisions, by allowing software engineers to select the derivation scenario to construct a new product based on their own preferences.

☞ In **Chapter 7** we demonstrate how we support software engineers in achieving the derivation of new product variants based on selection factors which are developer preferences and cost estimation.

3.3 Software Product Line Evolution

According to Botterweck and Pleuss [BP14], SPL evolution is complex due to the variability and the interdependencies between products. A new requirement or change in a requirement may affect several products. A change affects the whole product family and the related products and the SPL remains inconsistent until change is propagated. Moreover, a large interdependency exists between assets. Therefore, SPL evolution must be addressed in a systematic way. An SPL must evolve to reflect the new and changing requirements. Therefore, the more products it derives, the more expensive it becomes.

During SPL evolution, different abstraction levels must be taken into consideration [SE08]: *(1)* common assets defined at the SPL level, being part of all products, *(2)* variable assets that are part of some products and there contribution in a product is based on a variability decision and *(3)* product-specific assets that are part of an individual product and are not made for reuse. During evolution, assets can be added, modified and deleted. Therefore, any change can affect different products at different abstraction levels. Moreover, some assets may move from an abstraction level to another, most-likely from common to variable assets. For instance, once a new product is added and not using a common asset in its implementation, this asset becomes a variable asset. Furthermore, during their evolution, SPLs can be merged if they become similar over time [SE08], or an SPL can be splitted when parts of the SPL can evolve in different directions [SB99].

Botterweck and Pleuss [BP14] summarized SPL and product evolution strategies and situations presented in [SV02] and [DSB05] as:

— **Proactive evolution:** the proactive adoption approach, as we presented earlier, consists on proactively planning and adding requirements to the SPL during domain engineering activity. Since this approach integrates the current and expected customer requirements that belong to the scope of the SPL, it is considered as an evolution approach that can deal with market changes.

— **Reactive evolution:** reactive evolution is achieved by directly integrating in the SPL the new requirements that arise during product derivation. This approach is mainly adopted by model-driven SPLs [CAK+05], in order to avoid product-specific implementations. This allows a complete derivation of products from the SPL and an immediate reuse of the newly integrated requirements in future products derivation. However, this approach requires high efforts to ensure a co-evolution of existing products in parallel with the SPL evolution.

— **Branch-and-unite:** the branch-and-unite approach is related to the grow-and-prune concept [FV03]. It consists on creating a new product branch to deal with product-specific requirements before reunifying the branch(es) with the SPL after releasing the product.

— **Bulk:** the bulk situation occurs when a company creates several branches to evolve a product. This makes the reintegration of those branches in the SPL expensive.

— **Maintenance:** over time maintenance activities are performed such as bug fixing and refactoring at both SPL and product levels.

OBJECTIVE 15:
REACTIVE SPL EVOLUTION

In Objective 6, we highlighted on the need of an extractive SPL adoption approach when migrating existing product variants into an SPL. Furthermore, we are interested in a reactive SPL evolution, to offer an automated integration of the newly derived products in the SPL.

As per Mitschke *et al.* [ME08] traceability facilitates the maintenance and evolution of SPLs. They focused in their work on the traceability of the evolution of the relationships between artifacts, associated features and desired products, by assigning versions to each of them. New versions are created when changing in requirements occur: *"changes of the feature model can directly influence the implementation of features as well as products"*. Each version of an artifact is associated with a specific version of a feature and management of features dependencies. Therefore, traceability is required, to ensure that the SPL remains consistent after changes.

OBJECTIVE 16:
INCREMENTAL TRACEABILITY BETWEEN FEATURES AND ASSETS

We are interested in providing an incremental knowledge of the mappings between features and assets all along the evolution of the constructed SPL, in order to ensure on a one side that existing products are always available and derivable from the SPL and on the other side, keep mappings between the SPL artifacts up-to-date to support the derivation of new products with correct indicators.

☞ We demonstrate the incremental evolution of the correlations in **Chapter 7** and we evaluate it in **Chapter 9**.

According to Svahnberg and Bosch [SB99], the arising new products to derive may introduce conflicts with the existing SPL products. Therefore, they suggested a set of guidance in order to perform a controlled evolution of an SPL. Gomaa and Hussein [GH04] proposed software reconfiguration patterns to reconfigure the software architecture of an SPL to support a dynamic SPL evolution. Ajila and Kaba [AK08] suggested some SPL evolution mechanisms that focus on identifying the change, analyzing its impact, specifying its propagation and validating its functionality. They examined their mechanisms on three different evolution levels: architecture level, product line level and product level. Moreover, Hotz *et al.* [HWK06] proposed the *conIPF methodology*, which takes into consideration the occurring requirements during product configuration on application enginerring level, and allows their integration in the SPL and the domain engineering level. Similarily, Bayer *et al.* [BFK+99] presented the *PuLSE* framework, that allows not only SPLE, but also customizability of its components and maturity scale for structured evoluation and maintenance.

OBJECTIVE 17:
ALLOW A COMPLETE REUSE

We are interested in allowing a complete reuse of the SPL artifacts in a new product derivation, by allowing any possible configuration of features to be selected by the software engineer.

☞ We define in **Chapter 5** a constraint-free FM where all SPL features are optional except root feature. We call this FM a *free FM*, and its purpose is to enable complete reuse.

Feature models are considered as the key to manage SPL evolution due the abstraction that they provide [BP14]. Difference models are required to identify the changes that arise on the SPL feature model over time. Operators such as *add*, *modify*, *delete* and operators with richer semantics such as *split* are required to specify the operation performed on the feature model during a change. In the context of difference models, Schaefer *et al.* introduced the *Delta-Oriented Programming (DOP)* where they define an SPL as *(1)* a core module that specifies a valid product and *(2)* delta modules that specify changes to be applied to the core module in order to derive other products [SBB+10]. Moreover, Acher *et al.* proposed differencing techniques that exploit syntactic and semantic mechanisms in order to provide differences between feature models [AHC+12]. On the operators side, Alves *et al.* introduced feature model refactoring operations such as replacing a *mandatory* feature type with an *optional*, or converting an *or* to an *alternative* [AGM+06]. Such operations permit the evolution of an SPL, since it improves its configurability. As per Neves *et al.* SPL evolution might be risky, since it impacts existing products when introducing new features or improvements [NBA+15]. Therefore, they introduced safe evolution templates after analyzing different evolution scenarios. These templates can guide developers during the SPL evolution process in order to preserve the behavior of existing products. *EvoPL* is an approach that combines both difference models and change operators by using the abstraction provided by feature models as the main artifact to manage SPL evolution [BPPK09, BPD+10, PBD+12]. A feature model version is decomposed into model fragments. A fragment is composed of feature model elements that are added or removed together during an evolution step. The fragments and operators between their elements are stored in a specific feature model called *EvoFM*. A configuration of the *EvoFM* represents an evolution step, while the evolution of a FM is a set of configurations of the *EvoFM*. The approach proposed by Martinez *et al.* in [MZB+15b] does not provide an incremental evolution, since it does not offer an integration process of the newly derived products into the SPL.

OBJECTIVE 18:
MANAGING FEATURE MODEL EVOLUTION

In our approach, we are interested in updating the global feature model of the SPL whenever a new product is integrated in order to maintain systematic reuse.

☞ In **Chapter 7** whenever a new product is derived, we employ the *FAMILIAR merge* operation [ACLF13] to merge the SPL FM with the newly integrated product FM.

3.4 Summary and Contribution Choices

In this chapter, we presented the related works that propose approaches to migrate software product variants into an SPL, support the derivation of new products and evolve an SPL.

We have shown that an extractive SPL adoption must be performed in order to integrate PVs developed based on C&O into an SPL. In this context, in order to allow products derivation, it is important to identify the features implemented in the integrated PVs, map the features to their corresponding assets and construct the SPL FM. We presented several literature approaches to identify or locate features in PVs assets. In our dissertation, we consider that features must reflect the business functionalities of the SPL PVs. Moreover, we decided to focus on the integration of polyglot systems into an SPL. In this context, we aim to adopt a mechanism that takes into consideration the entire product line instead of treating products separately to identify the mappings between features and assets.

We presented several literature works that propose approaches to support the derivation of new products either in the SPL context or C&O context. Table 3.1 represents a comparison of the key characteristics offered by three of the main related works tools which are BUT4Reuse [MZB+17], ECCO [FLLHE14] and CACAO [LBC16]. Some approaches limit the software engineers ownership of PVs, since they automate the derivation, while others lack some operational level support. In our approach, we give interest in guide software engineers during the derivation process without imposing on them specific solutions. Therefore, we aim to propose the possible scenarios that they can rely on to derive a product, and keep decision-making on their behalf. The number of possible solutions might be large. Hence, to tackle complexity and facilitate decisions, we aim to allow software engineers to select the derivation scenario to construct a new product, based on their own preferences, i.e. by selecting the source of clone that is composed of the products that they are most familiar with. Moreover, we aim to cost-estimate the different proposed operations, so they can rely on it as an additional argument in their choice of the relevant operations to perform to achieve the derivation.

Finally, we presented different works in literature about the evolution of an SPL. We highlighted on the importance of adopting a reactive SPL evolution, in order to integrate the newly derived products in the SPL and benefit from their systematic reuse. Integrating new products involves new features and assets, hence, mappings between features and assets must be updated to take into consideration the arising changes. In our approach, we aim to provide an incremental evolution of the traceability between features and assets. Further, the integration of new products involves an update of the SPL FM. Moreover, we are interested in enabling a complete reuse, hence, providing software engineers with the ability to derive a new product implementing any set of features provided by the SPL. Figure 3.5 illustrates to which challenges the objectives of our approach are going to respond in the contribution part, which comes next.

Table 3.1: A comparison of the key characteristics of the main related work tools

		Tool		
		$BUT4Reuse$	$ECCO$	$CACAO$
Characteristics	Configuration through a feature model	Yes, configuration made through an SPL generated feature model	No, configuration made on the specification of the required features	No, configuration made based on the specification of new product requirements
	Mapping features to artifacts	Yes, provides both feature identification and feature location	Yes, provides feature location	Yes, detects which code methods are related to the new product requirements
	Derivation of new products	Yes, automated and dependent on the identified blocks of artifacts	Yes, automated derivation of new products with possibility of adding new features	Yes, manual derivation of new products with possibility of adding new requirements
	Support during derivation	No	Yes, provides hints to complete product	Yes, ranks legacy methods in the close legacy products according to the new product requirements
	Reuse of newly derived products	No, since feature model not updated to integrate them	Yes, since new products are integrated using an incremental mechanism	Yes, since relevancy check between requirements is done prior to each derivation

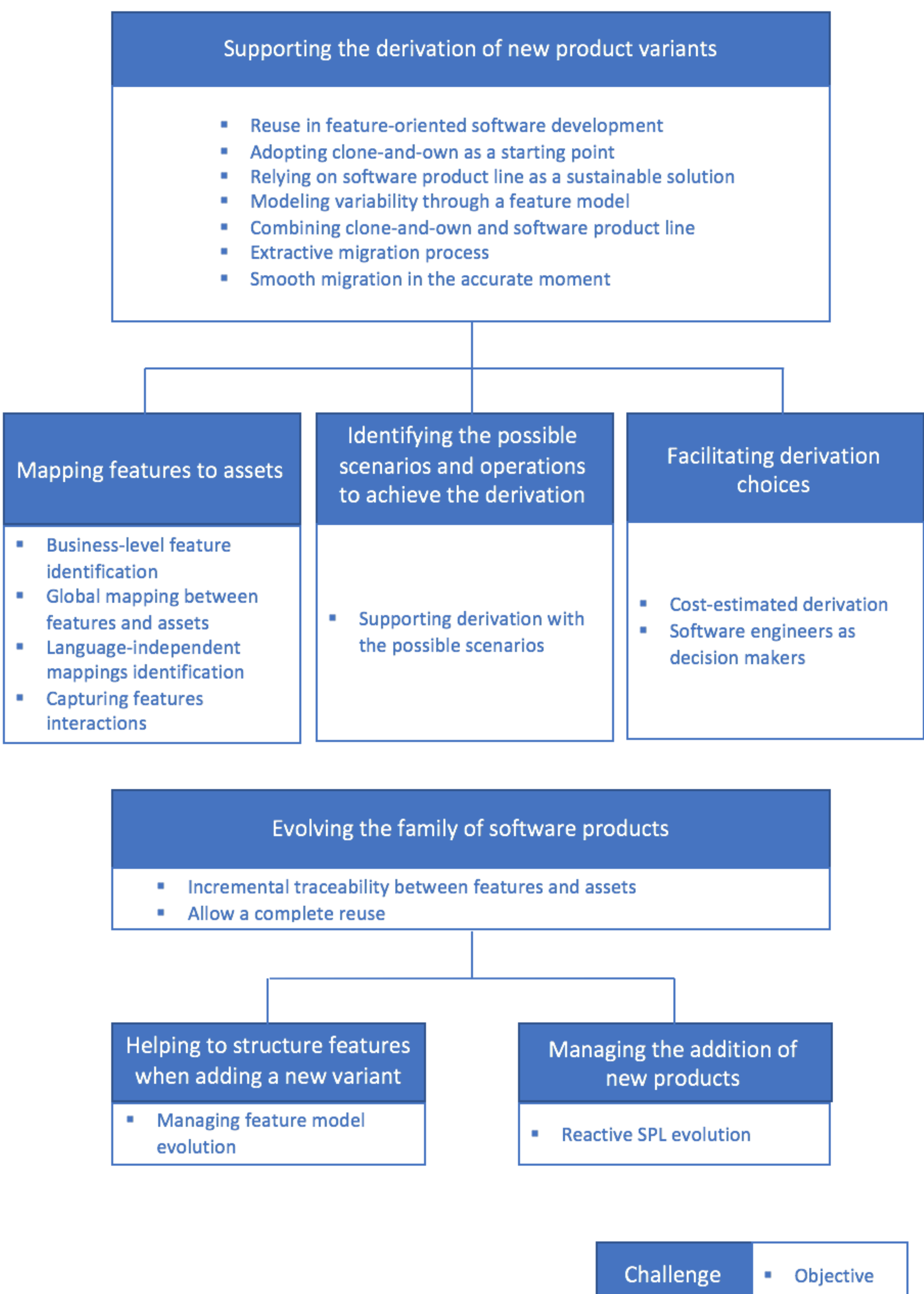


Figure 3.5: Challenges and objectives

Part II

Approach Contributions

In this part, we propose our approach to support Clone-and-Own in a Software Product Line context and we present its main contributions. Our approach consists first on migrating existing product variants into an SPL (see **Chapter 4**). Hence, an automated derivation of those products can be achieved, whenever a configuration requires a set of features realized by one of them. Further, new features can be added on top of the derived product. To provide an effective reuse, we support the derivation of new variants from the existing ones. Therefore, whenever the required features are subset of an existing one, or spread on several ones, comprising new features or not, we propose the possible scenarios to achieve the derivation. More precisely, we identify the products that can be the source for reuse, the required assets that have to be cloned from those products, and the operations to perform on the cloned assets to construct the new product (see **Chapter 5**). Such operations specify i.e. the features to be added on or removed from the cloned asset. To determine those operations, we rely on the automated mapping identification between the SPL features and assets, that we perform after the migration of the PVs (see **Chapter 4**). We estimate the cost of the possible scenarios and operations to facilitate the choice between them (see **Chapter 6**), and we provide a constraints system to simply allow end users to make their choices based on their own preferences (see **Chapter 7**). Hence, in our approach, we provide a complete guidance to end users to perform the derivation, that might involve different actors, such as product architects at configuration level and software developers at development level. Thus, product architects can select the appropriate derivation scenario and software developers can construct the product according to the proposed operations. Finally, our approach allows the integration of the newly derived products in the SPL to benefit from their reuse in future derivations, as well as preserving the derivation of existing products (see **Chapter 7**). Figure 3.6 shows our approach overview.

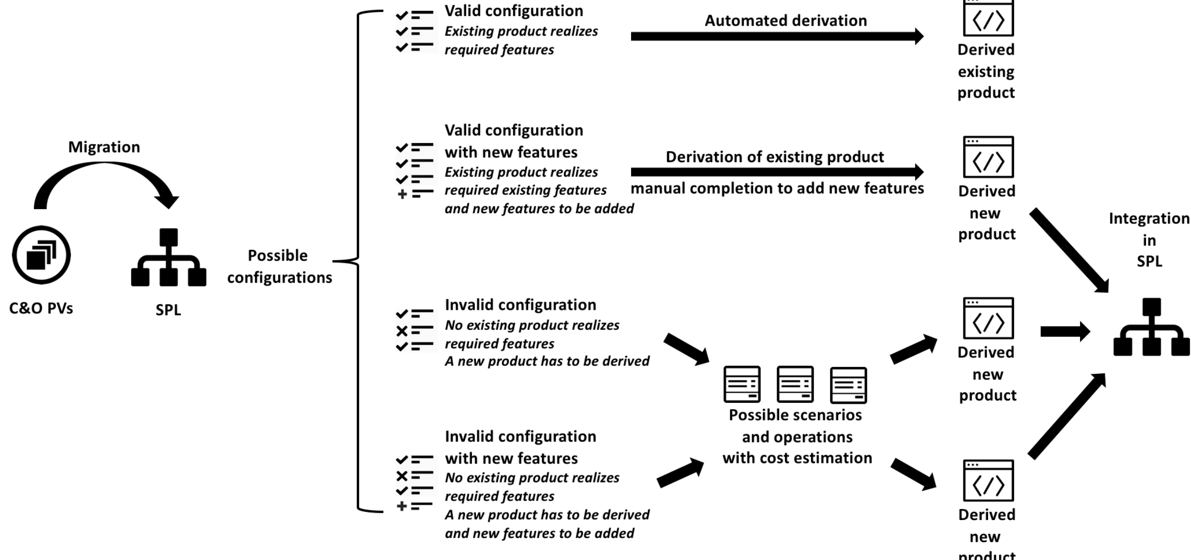


Figure 3.6: Approach overview

CHAPTER 4

MIGRATION PROCESS

Contents

4.1 Introduction **56**
4.2 Product Line Definition **58**
4.2.1 Feature Model Generation 58
4.2.2 Products and Assets Extraction and Storage 59
4.3 Correlations Identification **63**
4.4 Product Line Validation **68**
4.5 Product Line Limitations **69**
4.6 Summary **71**

4.1 Introduction

In this chapter, we address **Challenge 1.a** that we presented in the introduction of this dissertation. This challenge indicates, when developing a family of software products, the necessity of mapping the features of PVs to the assets implementing them. We highlighted in **Chapter 3** on the importance of managing variability by identifying and categorizing the features implemented by the PVs of the family of software products. This step is supposed to allow for example to determine what are the products that implement a set of required features, or if there exists a product that implements exactly all and no more than the required features. Thus, it is a step-forward to allow reuse of existing PVs. Another important issue consists in establishing mappings between features and assets. When the assets that are used to implement a certain feature are identified, it becomes possible to allow their reuse when constructing a new PV. We respond to **Challenge 1.a** by achieving the following objectives:

In this dissertation, we focus on feature-oriented software development where software reuse is practically performed using C&O or SPLE (**Objective 1**).

C&O is a practice adopted by many organizations to develop a family of software products since it offers rapidity, simplicity and independence [DRB+13]. Despite that C&O is considered as an efficient reuse practice, it loses its efficiency when the number of managed PVs becomes enormous, because it lacks a systematic methodology for reuse [FLLHE14].

A successful alternative to C&O is the adoption of an SPL development approach. When adopting an SPL approach, variability is managed efficiently, since features are categorized into common and variable features, and dependencies and constraints between them are identified [PBV05]. Variability is often expressed using a feature model [LKL02]. A configuration of an FM allows to simply recognize which products implement a set of required features, and reuse existing products whenever the required features are entirely implemented by one of them. Moreover, the development architecture adopted in SPLE consists of creating reusable assets in domain engineering [CN01], which enables reuse not only at product level but also at assets level.

Despite that SPLs allow to manage variability and enable reuse, they are considered as an expensive up-front investment [PBV05]. In practice, artifacts have to be defined and developed in domain engineering before being explored in application engineering to derive products. Many are the organizations that cannot take such an expensive investment in terms of resources and time at the early beginning of the family of products development, therefore, they tend instead to adopt simple practices such as C&O.

In this dissertation, we propose a hybrid approach for developing and managing a family of software products, in which we aim to explore the benefits provided by both C&O and SPLs. We give our attention to organizations that adopt C&O as their initial development approach. Since such organizations are used to or convinced in adopting C&O, we give interest on retaining the fact that PVs are constructed by cloning (**Objective 2**). On the other hand, since in C&O variability is not managed and mappings are not established, we give interest in adopting SPL (**Objective 3**). Hence, as an initial step in our approach, we consider migrating PVs developed based on C&O into an SPL in order to manage their variability and facilitate their reuse (**Objective 5**). We perform this migration through an extractive approach (**Objective 6**).

We are interested in constructing an FM to allow the configuration of products by selecting the required features (**Objective 4**). Moreover, features must reflect the business-level functionalities implemented by the SPL products. Thus, in our approach we assume that a feature model representing the features implemented by each product variant is assigned to it, prior to its migration into the SPL (**Objective 7**).

We propose an automated mappings discovery between the features and the assets of the constructed SPL (**Objective 8**). Our approach takes into consideration the entire product line instead of treating products separately to identify the mappings. We call the mappings *"correlations"* since they define the correlation level between the artifacts of the SPL. Since we are interested in polyglot systems, the mappings discovery mechanism that we propose is language-independent (**Objective 9**). We highlighted earlier on the fact that a feature might have different implementations due to features interactions. Moreover, a many-to-many relationship between features and assets is likely to occur. Hence, we take feature interactions into account when setting up the mappings discovery mechanism (**Objective 10**).

Finally, we assume that once the family of products starts to have its very few products, it becomes favorable to migrate them into the SPL (**Objective 11**). We do not impose this criteria, however, we consider it as a recommendation. Meanwhile, we guarantee in our approach a smooth and quick migration process that do not affect the development productivity.

Figure 4.1 summarizes the migration process of our approach. Existing PVs are migrated into an SPL by means of their provided artifacts and feature models. The SPL feature model is generated and correlations (mappings) between features and assets are identified. The generated FM allows the automated derivation of the integrated PVs.

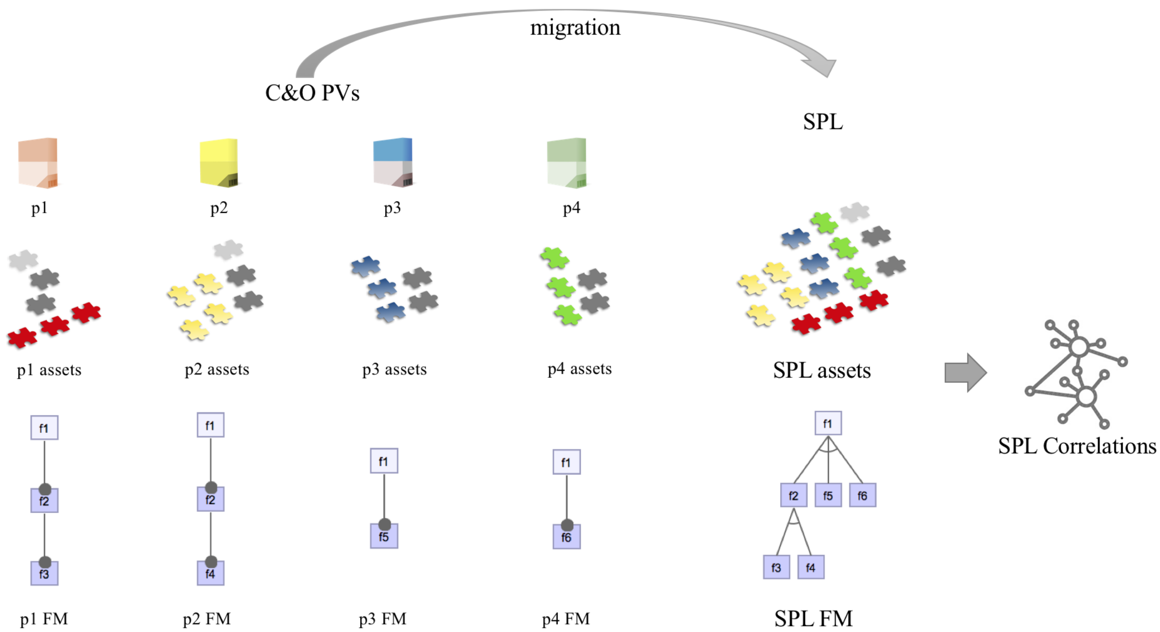


Figure 4.1: Migration process

We define the migration process of our approach as an extractive process that is characterized by the following activities:

1. Structuring the SPL
   (a) Generating the SPL feature model
   (b) Extracting and storing products and assets
2. Identifying the correlations between features and assets

4.2 Product Line Definition

Migrating PVs into an SPL consists in providing the implementation files of the PVs and specifying the features that they implement. In our approach, we consider that features of a PV are known prior to migration process. We make this choice since we consider that a feature must represent a business-level functionality.

As a general definition, we define a *Software Product Line* $\mathcal{SPL}$ as a family of polyglot software products that share a set of identified features implemented through a set of assets. Our definition of an $\mathcal{SPL}$ fits into the classic SPL definition of Clements and Northrop [CN01]. In our approach, we aim to manage the set of features shared between the software products, and identify the assets that implement them.

4.2.1 Feature Model Generation

In order to manage the features of the $\mathcal{SPL}$, and since we aim to deal with business-level features, we consider that, for each software product, the features that it implements are given. In addition, prior to the migration of the software products, our approach imposes the representation of the features implemented by each of them in the form of a feature model. We impose this requirement, in order to respect the coherence of the overall structure of the features implemented by the $\mathcal{SPL}$. In this context, an approach proposed by Ziadi *et al.* [ZFdSZ12, ZHP+14, Mar16] can be used to automatically identify the migrated products features and generate the SPL FM, in situations where feature identification is required.

Definition 4.2.1 (Feature) :
We define a *feature* f, identified by its $name$, as an abstraction of a business functionality.

Definition 4.2.2 (Feature Model) :
We define a *feature model* fm as a triple $\langle id, \{f_1, ...f_n\}, struct\rangle$, where id is the identifier of the feature model, $\{f_1, ...f_n\}$ is the set of features that belong to it, and $struct$ is the structure of the feature model, representing in a tree structural format the features and the constraints between them. We note the set of features $F(fm) = \{f_1, ...f_n\}$.

In our approach, the structure of a feature model respects the notations and semantics [Bat05, BSRC10] of a classic feature model that was first introduced in the FODA method [KCH+90].

Given a product p, its feature model has no variability, since it represents the configuration of the product which represents simply the features that it implements.

Example 1: Product variants implementation files and feature models

The running example PVs with an excerpt of their corresponding implementation files and the FMs representing the features they implement are shown in Table 4.1.

Table 4.1: Running example product variants with their implementation files and feature models

Product	File$^{\text{version}}$	Feature Model
p_1	match.jsp^1 SaveMatch.java^1 style.css^1	ManageMatches AddMatches, ModifyMatches
p_2	match.jsp^1 SaveMatch.java^1 style.css^2 DeleteMatch.java^1	ManageMatches AddMatches, ModifyMatches, DeleteMatches
p_3	match.jsp^1 SaveMatch.java^2 style.css^3	ManageMatches AddMatches

A global FM represents the collection of features provided by the $\mathcal{SPL}$ and the constraints between them. Indeed, a configuration via this FM allows the derivation of the migrated PVs. In our approach, we employ the *merge* operation provided by *FAMILIAR* language [ACLF13] to construct the global FM of the $\mathcal{SPL}$. We apply the *merge* operation on the FMs of the PVs to obtain the $\mathcal{SPL}$ FM.

Definition 4.2.3 (Restrictive FM) :
We call the generated feature model a *restrictive FM*, since it restricts but also guarantees a valid configuration and an automated derivation of the exact set of products provided by the $\mathcal{SPL}$.

4.2.2 Products and Assets Extraction and Storage

We define the $\mathcal{SPL}$ products by means of the migrated PVs. For each PV an $\mathcal{SPL}$ product is defined. In order to identify the $\mathcal{SPL}$ assets, they must be extracted from the implementation artifacts of the PVs. Since PVs have features in common, they are supposed to have assets in common too. A feature might be implemented by several assets. In addition, an asset might share the implementation of several features. Consequently, the presence of a feature in a PV and its absence in another PV often produces two versions of a same asset in two different PVs. Thus, for some assets, several versions can be identified in the migrated PVs.

In our approach, we perform language independent feature location, in contrary to most approaches in literature [AM14, FLLHE14, Mar16], that perform a language-specific feature location, by employing artifact-type based algorithms to extract assets. In polyglot PVs, files of different formats exist. In addition, some files might be written using several languages. For instance, a *Java Server Page (JSP)* file, such as $match.jsp$ form the running example (see

Table 1.2), might contain *HTML*, *Java*, *CSS* blocks of code, and others. Thus, employing language-specific feature location techniques for polyglot PVs might be ineffective and error-prone.

Example 2: Applying the FAMILIAR merge operation to construct the $\mathcal{SPL}$ FM

Below are the FMs of the running example PVs in *FAMILIAR* language. A *FAMILIAR* merge operation generates the $\mathcal{SPL}$ FM.

Listing 4.1: Running example PVs FMs in FAMILIAR language

```
fm_p1 = FM (ManageMatches: AddMatches ModifyMatches;)

fm_p2 = FM (ManageMatches: AddMatches ModifyMatches DeleteMatches;)

fm_p3 = FM (ManageMatches: AddMatches;)
```

Listing 4.2: FAMILIAR merge operation over PVs FMs

```
fm_spl = merge sunion fm*
```

Listing 4.3: $\mathcal{SPL}$ FM generated from FAMILIAR merge operation

```
fm_spl: (FEATURE_MODEL) ManageMatches: AddMatches [ModifyMatches] [DeleteMatches];
(DeleteMatches -> ModifyMatches);
```

As shown in the figure below, the feature $ManageMatches$ is the root feature. The feature $AddMatches$ is mandatory, since it is implemented by all PVs, while the features $ModifyMatches$ and $DeleteMatches$ are optional since they are implemented in some but not all PVs. The constraint $DeleteMatches \Rightarrow ModifyMatches$ is identified since each product implementing the feature $DeleteMatches$ implements also the feature $ModifyMatches$.

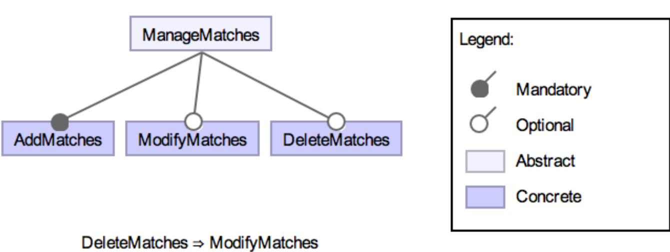


Figure 4.2: Running example $\mathcal{SPL}$ global FM

Since we are interested in polyglot PVs, we set the granularity of the assets to be identified at file level. In other words, each extracted file from a PV corresponds to an asset. Since several versions of a file might be extracted, several versions of an asset might exist. Therefore, when extracting the assets, we distinguish different versions, and for each product we identify which version was used. Thus, we call *assets* the identified files and *asset instances* their corresponding versions. An asset represents an abstraction of one or more files having the same name in one or more PVs, noted as instance(s) of the asset.

Definition 4.2.4 (Asset Instance) :
We define an *asset instance* a^{i} of an asset a as a pair $\langle instanceNo, implementation\rangle$, where $instanceNo$ is the instance number (version) of the asset instance, and $implementation$ is its corresponding implementation file.

A new asset instance of an asset is identified, when a file existing in another product has the same name with a different content.

Definition 4.2.5 (Asset) :
We define an *asset* a as a pair $\langle name, \{a^1, ..., a^{\mathrm{n}}\}\rangle$, where $name$ is the name of an implementation file and $\{a^1, ..., a^{\mathrm{n}}\}$ is the set of instances of the asset a, which are the different implementation versions of the file. We note $AI(a) = \{a^1, ..., a^{\mathrm{n}}\}$.

Asset name and file name: We designate by *asset name* and *file name*, the relative path of the file in concern, including its exact name, within the project that it belongs to. However, for simplicity and to make the examples comprehensive, we represent files and assets throughout the dissertation by their exact name excluding the path. For example, in our approach $match.jsp$ refers to $WebContent/match/match.jsp$.

Definition 4.2.6 (Product) :
We define a *product* p as a triple $\langle name, fm, \{a^i_j, ..., a^n_m\}\rangle$. A product p is identified by its $name$, a feature model fm referencing the features that it *implements* and a set of asset instances that it *exploits* to fulfill its implementation. We note $fm(p) = fm$ to refer to the feature model of p, and $AI(p) = \{a^i_j, ..., a^n_m\}$ to refer to the asset instances exploited by p.

The products of the $\mathcal{SPL}$ are defined by means of the migrated PVs. We consider products as the main elements of an $\mathcal{SPL}$, since they embody the features and the assets of the SPL. Products are founded by their asset instances and their relationship with assets are identified via their asset instances.

Definition 4.2.7 (Product implements features) :
Given a product p, p *implements* a set of features noted as $F(p) = F(fm(p))$.

We can now define formally an $\mathcal{SPL}$ as follows:

Definition 4.2.8 (Software Product Line $\mathcal{SPL}$) :
We define a *Software Product Line* $\mathcal{SPL}$ as a set of products $\{p_1, ..., p_n\}$.

- We note the $\mathcal{SPL}$ products $P(\mathcal{SPL}) = \{p_1, ..., p_n\}$.
- We note the $\mathcal{SPL}$ features $F(\mathcal{SPL}) = \cup_{p_j \in P(\mathcal{SPL})}(F(p_j)) = \{f_1, ..., f_x\}$.
- We note the $\mathcal{SPL}$ assets $A(\mathcal{SPL}) = \{a \mid \forall a^i \in \cup_{p_j \in P(\mathcal{SPL})}(AI(p_j)), a^i \in AI(a)\} = \{a_1, ..., a_z\}$.

As we work only with one $\mathcal{SPL}$, we denote $P(\mathcal{SPL})$ as $\mathcal{P}$, $F(\mathcal{SPL})$ as $\mathcal{F}$ and $A(\mathcal{SPL})$ as $\mathcal{A}$.

Definition 4.2.9 (Artifact) :
An artifact is used in the development of software systems. In our approach we limit its definition to the following: an *artifact* is a *feature*, an *asset* or an *asset instance* participating in the development of $\mathcal{SPL}$.

Definition 4.2.10 (Product employs assets and exploits asset instances) :
Given a product p, p *employs* a set of assets noted as $A(p)$ and for each asset $a_j \in A(p)$, p *exploits* one of its instances $a_j^i \in AI(a_j)$ to fulfill the implementation where $A(p) = \{a_j \mid a_j \in \mathcal{A}, \exists a_j^i \in AI(p), a_j^i \in AI(a)\}$.

Definition 4.2.11 (Feature implemented by products) :
Given a feature f, the set of products that f is implemented in is noted as $P(f) = \{p \in \mathcal{P}, f \in F(p)\}$, where $P(f) \subseteq \mathcal{P}$.

Definition 4.2.12 (Asset employed by products) :
Given an asset a, the set of products that a is *employed* by is noted as $P(a) = \{p \in \mathcal{P}, a \in A(p)\}$, where $P(a) \subseteq \mathcal{P}$.

Definition 4.2.13 (Asset instance exploited by products) :
Given an asset instance a^{i} the set of products that a^{i} is *exploited by* is noted as $P(a^{\text{i}}) = \{p \in \mathcal{P}, a^{\text{i}} \in AI(p)\}$, where $P(a^{\text{i}}) \subseteq \mathcal{P}$.

Property 4.2.1 ($\mathcal{SPL}$ facts) :
Given a product p, an asset a and a feature f, we can demonstrate that:

— $a \in A(p) \Leftrightarrow p \in P(a)$

— $f \in F(p) \Leftrightarrow p \in P(f)$

Figure 4.3 shows the $\mathcal{SPL}$ model diagram. Algorithm 1 in **Chapter 8** explains how the assets and their instances are extracted from the migrated PVs implementation files, and how the $\mathcal{SPL}$ products are identified.

Property 4.2.2 (Asset instance uniqueness in product) :
Since an asset is identified based on a file path, given a product $p \in \mathcal{P}$, for each asset a employed by p, only one of its instances is exploited by p. A product verifies uniqueness of asset instances if $\forall a \in A(p), \exists! a^{\text{i}} \in AI(p), a^{\text{i}} \in AI(a)$.

Example 3: Running example extracted assets and instances

In the running example PVs, the $match.jsp$ asset represents a “Java Server Page” that displays a match information. The same implementation of the corresponding file is used in all PVs. Thus, only 1 instance was identified for it. On the other hand, the asset $style.css$ represents a style sheet file that is employed by several web pages. Therefore, different implementations of this asset are available. Thus, 3 instances were identified for it. Table 4.2 shows the assets and corresponding instances, identified in the products of the $\mathcal{SPL}$.

Table 4.2: Running example relationships between assets, asset instances and product variants

Asset	Instance	Product		
		p_1	p_2	p_3
match.jsp	1	✓	✓	✓
SaveMatch.java	1	✓	✓	
	2			✓
style.css	1	✓		
	2		✓	
	3			✓
DeleteMatch.java	1		✓	

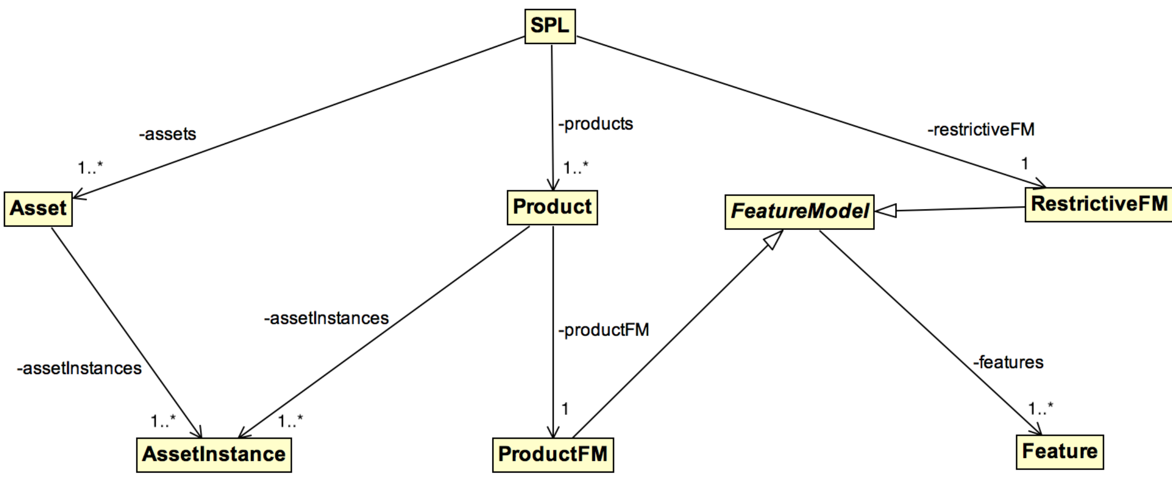


Figure 4.3: $\mathcal{SPL}$ model diagram

4.3 Correlations Identification

In order to identify the mappings among the $\mathcal{SPL}$ artifacts, we define *"correlations"*. A correlation[1] indicates the coexistence between a feature and an asset, or between a feature and an asset instance. An asset is a global abstraction of a partial or a total implementation of one or more features, since a feature implementation can be spread into several assets, and similarly, an

[1]Correlation: a mutual relationship or connection between two or more things (Oxford Dictionary)

asset can include implementation fragments of different features. Therefore, we are interested in identifying the correlation between a feature and an asset, in order to determine the coexistence between them in the $\mathcal{SPL}$ products. On the other hand, an asset instance realizes one of the asset implementations in one or more $\mathcal{SPL}$ products. Therefore, we are interested in identifying the correlation between a feature and an asset instance.

In our approach, instead of mapping a feature or set of features (features interaction) to an implementation block, which can be composed of fragments of several assets, we map each feature to the set of assets that supposedly contribute in its implementation. We call those mappings *correlations*. Hence, a feature might be correlated to several assets, and an asset might be correlated to several features as well.

Definition 4.3.1 (Correlation) :
We designate the $\mathcal{SPL}$ correlations $C(\mathcal{SPL})$ noted as $\mathcal{C}$ as the set of correlations that handles the coexistence between features and assets, and between features and asset instances in the $\mathcal{SPL}$ products. Therefore, we define two subsets of correlations:

1. *Feature to Asset* correlations, and

2. *Feature to Asset Instance* correlations.

A correlation between a feature f and an asset a holds if the following constraints are valid:

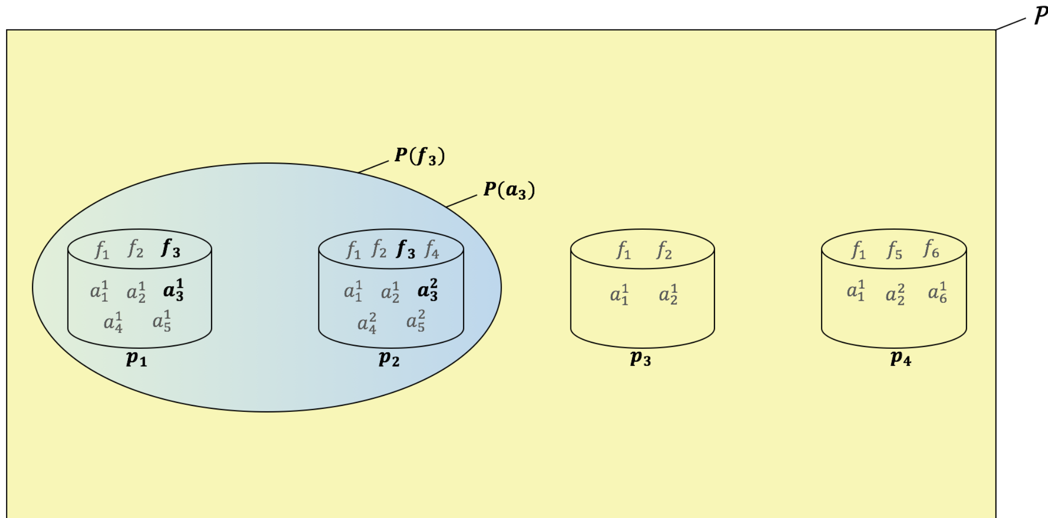
— For each product p_j that implements f, p_j employs a (exploits any of its instances)

— There does not exist an asset instance a^{i} exploited by p_j and by another product p_k that does not implement f

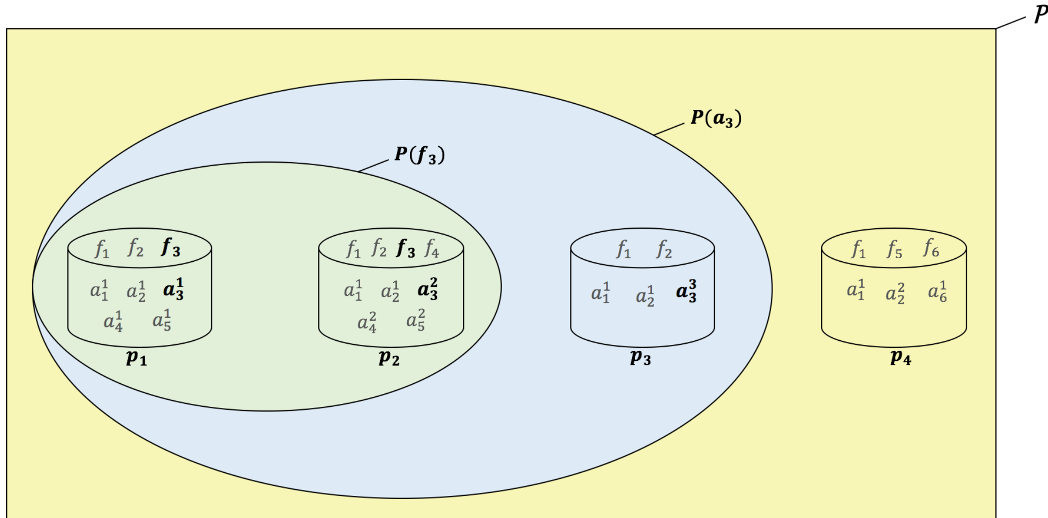
Definition 4.3.2 (Feature to Asset Correlation) :
A correlation between a feature f and an asset a noted as $c(f, a) \in \mathcal{C}$ holds, if $P(f) \subset P(a) \wedge \forall a^{\mathrm{i}} \in AI(P(f)), a^{\mathrm{i}} \notin AI(P(a) \setminus P(f))$

Definition 4.3.3 (Correlation Type) :
We define two correlation types *(CT)*: $\{equivalence, implication\}$, where the type t of a correlation is noted as $t(c(f, a))$. A correlation $c(f, a)$ is an *equivalence* (noted $f \Leftrightarrow a$) if $P(f) = P(a)$, otherwise it is an *implication* (noted $f \Rightarrow a$).

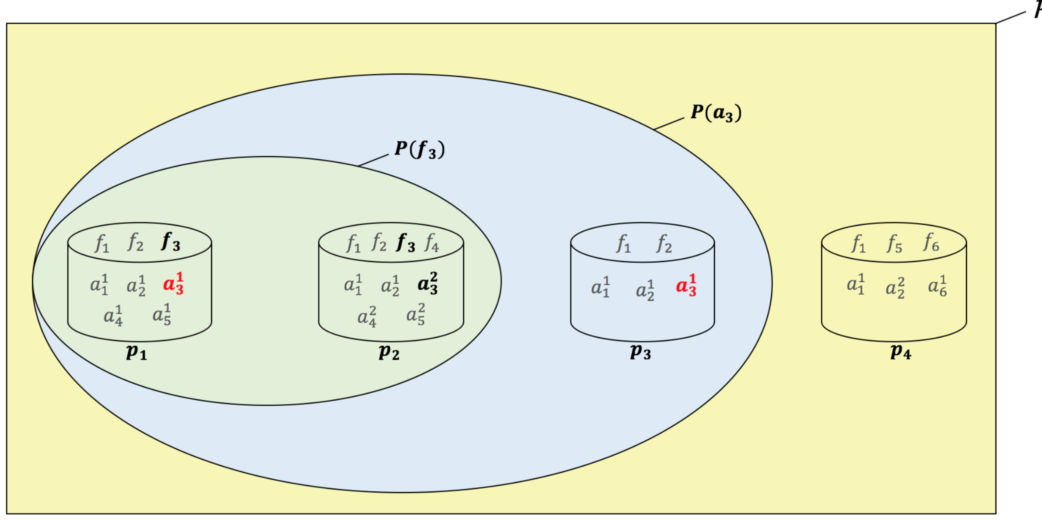
In Figure 4.4, we demonstrate when a feature to asset correlation is identified. Given a feature f_3 and an asset a_3, in the first situation (see Figure 4.4a), there exists an equivalence correlation $f_3 \Leftrightarrow a_3$, since p_1 and p_2 are the only products that implement f_3 and employ a_3. In the second situation (see Figure 4.4b), there is only an implication correlation $f_3 \Rightarrow a_3$, since p_1 and p_2 implement f_3 and employ a_3, while there exists a product p_3 that employs a_3 without implementing f_3. In the third situation (see Figure 4.4c), no correlation holds between f_3 and a_3, because p_3 does not implement f_3 and exploits the same instance a_3^1 of a_3 exploited by p_1 that implements f_3.



(a) Equivalence correlation between f_3 and a_3



(b) Implication correlation between f_3 and a_3



(c) No correlation between f_3 and a_3

Figure 4.4: Examples of feature to asset correlation

Example 4: Running example feature to asset correlations

Table 4.3 shows the correlations between the features and the assets of the running example. If we refer to Table 1.1 and Table 1.2, we can see that the three products implement the feature $ManageMatches$ and employ the asset $match.jsp$, therefore, $ManageMatches \Leftrightarrow match.jsp$. On the contrary, for each product implementing the feature $ModifyMatches$, which are p_1 and p_2, the asset $SaveMatch.java$ is employed, but p_3 employs $SaveMatch.java$ and does not implement the feature $ModifyMatches$. Therefore, there does not exist an equivalence correlation between $ModifyMatches$ and $SaveMatch.java$. However, $ModifyMatches \Rightarrow SaveMatch.java$ because p_3 does not exploit the same instance of $SaveMatch.java$ exploited by p_1 and p_2. As a counter example, there is no implication between $ModifyMatches$ and $match.jsp$, since p_3 that does not implement $ModifyMatches$ exploits the same instance of $match.jsp$ exploited by both p_1 and p_2 that implement $ModifyMatches$.

Table 4.3: Correlations between features and assets of the running example

Feature	CT	Asset
ManageMatches	$\Leftrightarrow$	match.jsp
AddMatches	$\Leftrightarrow$	match.jsp
ManageMatches	$\Leftrightarrow$	SaveMatch.java
AddMatches	$\Leftrightarrow$	SaveMatch.java
ModifyMatches	$\Rightarrow$	SaveMatch.java
ManageMatches	$\Leftrightarrow$	style.css
AddMatches	$\Leftrightarrow$	style.css
ModifyMatches	$\Rightarrow$	style.css
DeleteMatches	$\Rightarrow$	style.css
DeleteMatches	$\Leftrightarrow$	DeleteMatch.java

Figure 4.5 shows the correlations model diagram.

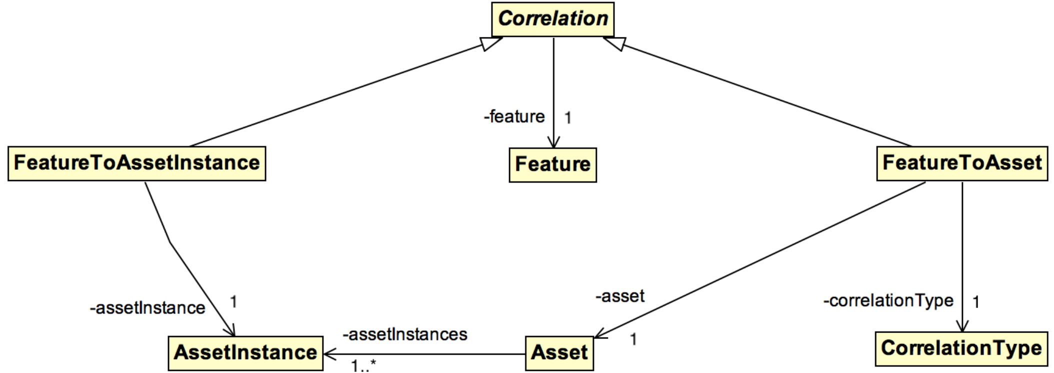


Figure 4.5: Correlations model diagram

Definition 4.3.4 (Feature to Asset Instance Correlation) :
Given an instance a^{i} of an asset a, a correlation between a feature f and a^{i} noted as $c(f, a^{\mathrm{i}})$ holds if $c(f, a) \wedge \exists p \in P(f), a^{\mathrm{i}} \in AI(p)$.

Example 5: Running example feature to asset instance correlations

As shown in Example 4, the correlation $c(ModifyMatches, SaveMatch)$ holds. Consequently, the correlation $c(ModifyMatches, SaveMatch^1)$ holds, since $SaveMatch^1$ is exploited by p_1 and p_2 that implement $ModifyMatches$. In contrary, the correlation $c(ModifyMatches, SaveMatch^2)$ does not hold, since the product p_3 exploiting $SaveMatch^2$ does not implement $ModifyMatches$. The correlations between the features and the asset instances of the running example are shown in Table 4.4.

Table 4.4: Correlations between features and asset instances of the running example

Feature	**Asset$^{\text{instance}}$**
ManageMatches	match.jsp^1
AddMatches	match.jsp^1
ManageMatches	SaveMatch.java^1
AddMatches	SaveMatch.java^1
ModifyMatches	SaveMatch.java^1
ManageMatches	SaveMatch.java^2
AddMatches	SaveMatch.java^2
ManageMatches	style.css^1
AddMatches	style.css^1
ModifyMatches	style.css^1
ManageMatches	style.css^2
AddMatches	style.css^2
ModifyMatches	style.css^2
DeleteMatches	style.css^2
ManageMatches	style.css^3
AddMatches	style.css^3
DeleteMatches	DeleteMatch.java^1

Definition 4.3.5 (Asset correlated features) :
Given an asset a, the set of features that a is in correlation with is noted as $F(a) = \{f \mid f \in \mathcal{F}, c(f, a)\}$.

Definition 4.3.6 (Asset instance correlated features) :
Given an instance a^{i} of an asset a, the set of features that a^{i} is in correlation with is noted as $F(a^{\text{i}}) = \{f \mid f \in \mathcal{F}, c(f, a^{\text{i}})\}$.

Definition 4.3.7 (Feature correlated assets) :
Given a feature f, the set of assets that f is in correlation with is noted as $A(f) = \{a \mid a \in \mathcal{A}, c(f, a)\}$.

Definition 4.3.8 (Feature correlated asset instances) :
Given a feature f and an asset a_j, the set of instances of a_j that f is in correlation with is noted as $AI(f/a_j) = \{a_j^i \mid a_j^i \in AI(a_j), c(f, a_j^i)\}$. The set of instances of all assets that f is in

correlation with is noted as $AI(f) = \cup_{a_j \in A(f)}(AI(f/a_j))$.

4.4 Product Line Validation

In this section, we identify some structural requirements that must be satisfied during migration process in order to create the $\mathcal{SPL}$.

As defined earlier, a product is identified by its name. However, when migrating PVs, a software engineer might attempt to provide two products having the same name. Thus, this attempt is considered as an anomaly that must be detected and rejected.

R1: each product in $\mathcal{SPL}$ must have a unique name, where $\forall p_j \in \mathcal{P}, \nexists\, p_k \in \mathcal{P} \mid name(p_j) = name(p_k)$.

When a PV is migrated into the $\mathcal{SPL}$, an FM representing the features that it implements is assigned to it. In our approach, we employ the union merge operation provided by FAMILIAR, in order to generate the restrictive fm. FAMILIAR imposes that the root features of the input feature models match as a rule to accomplish the operation [Ach11]. Therefore, the provided PVs FMs must have a proper structure where all of them have the same root feature.

R2: each product must have a feature model respecting the FODA notations and semantics and all products have to share at least one common feature which is the root feature of the $\mathcal{SPL}$, where $\forall (p_j, p_k) \in \mathcal{P}, \exists\, f_{root} \mid f_{root} \in \mathcal{F} \wedge f_{root} \in F(p_j) \wedge f_{root} \in F(p_k)$.

In our approach, we consider that products must be distinguished at business-level. Therefore, no two products are allowed to implement exactly the same features.

R3: we prohibit that two products in the $\mathcal{SPL}$ implement exactly the same set of features, regardless if their implementation is identical or not. Thus, $\forall p_j \in \mathcal{P}, \nexists\, p_k \in \mathcal{P} \mid F(p_j) = F(p_k)$.

Since no two products are accepted if they implement the same features, respectively, no two products are supposed to have the same implementation. Furthermore, if two products implement different features, they are supposed to have different implementations (different asset instances). Therefore, in our approach, we enforce that no two products should have the same asset instances.

R4: we prohibit that two products in the $\mathcal{SPL}$ have exactly the same implementation. In other words, no two products should have exactly the same asset instances. Thus, given two products $(p_j, p_k) \in \mathcal{P}$ if they employ the same assets, they might exploit some, but definitely not all the same asset instances. Thus, if $A(p_j) = A(p_k) \Rightarrow AI(p_j) \neq AI(p_k)$.

Based on **R3** and **R4**, a product is accepted for migration if there does not exist another product that implements the same features or exploits exactly the same set of assets.

If a product p_j implements a set of features that is subset of the set of features implemented by another product p_k, and the assets employed by p_j are not part of the ones employed by

p_k, then each product is employing a different set of assets to implement the same features. Therefore, this is considered as an anomaly that produces uncorrelated artifacts.

R5: given two products $(p_j, p_k) \in \mathcal{P}$, if p_k implements all the features implemented by p_j and more, p_k must employ all the assets employed by p_j and – not necessarily but most likely – more. Thus, if $F(p_j) \subset F(p_k) \Rightarrow A(p_j) \subseteq A(p_k)$.

Property 4.4.1 (Complete $\mathcal{SPL}$) :
An $\mathcal{SPL}$ is *complete*, if all validation requirements are satisfied and each of its artifacts participates in at least one correlation.

In our approach, the migration process is accomplished if the $\mathcal{SPL}$ to be generated is a *complete* $\mathcal{SPL}$.

4.5 Product Line Limitations

Architecture as a keystone of the $\mathcal{SPL}$:
Extraction of assets, and respectively asset instances is done based on their corresponding file name and the path under which they are located in the PV(s) from which they are extracted. Therefore, a change in folder structure, or in the name of a file in a certain PV implies the identification of a new asset for the file in concern, which causes an inconsistency in the $\mathcal{SPL}$ structure and an identification of spurious correlations. Thus, the extraction of the $\mathcal{SPL}$ assets is dependent on the architecture of the PVs. Similarly, the features of the $\mathcal{SPL}$ are extracted from the PVs FMs. Therefore, any change in FMs affects the identified features, and respectively the $\mathcal{SPL}$ correlations.

Asset instance uniqueness:
Since assets are identified by the path and the name of the file to which they refer, and an asset instance is a version of the file existing in one or more PVs, then, it is not possible to have two instances of the same asset in a product. A product can exploit only one asset instance of a certain asset.

Theoretically identified correlations:
The identified correlations are discovered based on the theory that we defined in this chapter, by analyzing the variability between the PVs artifacts. Despite that we aim to reach the highest precision in the identified correlations, our approach does not guarantee that all identified correlations are certain, because variability between PVs might be identified by means of a set of features and not a single feature. For instance, if a set of features $CF = \{f_1, ..., f_n\}$ consists of the common features between all $\mathcal{SPL}$ products, and a set of assets $CA = \{a1, ..., a_m\}$ is the set of common assets between all $\mathcal{SPL}$ assets, our approach generates an equivalence correlation between each feature $f_j \in CF$ and each asset $a_j \in CA$, however, nothing guarantees in practice that each $f_j \in CF$ is in correlation with each asset $a_j \in CA$ and vice versa. Similarly, if a set of features $PF = \{f_1, ..., f_n\}$ is implemented by a product and not others, each asset a_j of

the set of asset $PA = \{a_1, ..., a_m\}$ employed to implement the features will be in correlation with each feature $f_j \in PF$, however, nothing guarantees this in practice. Despite that the correlations identified by our approach may not guarantee a high precision when few PVs are initially migrated, our approach ensures a continuous refinement of the correlations that gain precision over time with each integration of a new product in the $\mathcal{SPL}$.

Example 6: Example of uncorrelated asset if R5 violated

In this example, we illustrate a situation in which an asset has no correlations due to the violation of the requirement **R5**.

In this situation, we have two PVs, as shown in Table 4.5, p_x implements the features $welcome$ and $news$, and p_y implements the features $welcome$, $news$ and $contact$. Table 4.6 shows the assets and their corresponding instances exploited by the products.

Table 4.5: Product variants with their corresponding features

	welcome	news	contact
p_x	✓	✓	
p_y	✓	✓	✓

Table 4.6: Product variants with their corresponding asset instances

Product	**Asset ^{instance}**
p_x	index.html^1 news.html^1
p_y	home.html^1 news.html^1 contact.html^1

Table 4.7 shows the correlations between the features and the assets, and as shown, the asset $index.html$ has no correlations. This occurred because $index.html$ was not employed by p_y where we have $home.html$ instead. Since the features implemented by p_x are subset of the ones implemented by p_y, all assets employed by p_x were supposed to be employed by p_y. Apparently, both $index.html$ in p_x and $home.html$ in p_y implement the $welcome$ feature while they have different names. **R5** was set to detect such situations and prevent the construction of an *incomplete* $\mathcal{SPL}$.

Table 4.7: Correlations between features and assets

Feature	**CT**	**Asset**
welcome	$\Leftrightarrow$	news.html
news	$\Leftrightarrow$	news.html
contact	$\Leftrightarrow$	home.html
contact	$\Leftrightarrow$	contact.html

Ad-hoc algorithm to identify correlations:
In our approach, we defined our own ad-hoc algorithm to identify mappings instead of relying on formal classification algorithms such as Formal Concept Analysis (FCA). However, we relied on our algorithm because we are interested in identifying feature correlations at both asset and asset instance level. Moreover, the constraints imposed by our algorithm allow to identify less correlations compared to FCA algorithm, which gives a higher level of correctness to the $\mathcal{SPL}$ correlations.

4.6 Summary

In this chapter, we responded to the first challenge addressed in the dissertation. By migrating PVs into the $\mathcal{SPL}$, we enabled variability management and allowed the automated derivation of the products from the $\mathcal{SPL}$. In addition, we identified the mappings between the SPL artifacts in terms of correlations.

To allow the derivation of the migrated PVs from the $\mathcal{SPL}$, the *restrictive* FM is generated from the FMs of the migrated PVs. The *restrictive* FM permits the configuration of the $\mathcal{SPL}$ features in order to reuse the $\mathcal{SPL}$ products.

The identified correlations represent the mappings between the $\mathcal{SPL}$ features and assets and respectively between the features and the asset instances. These correlations are supposed to facilitate the reuse of $\mathcal{SPL}$ artifacts in order to create new products. We explain this process throughout the upcoming chapters.

The PVs to be migrated must respect some structural requirements, in order to make sure that their migration will result in a *complete* $\mathcal{SPL}$. A *complete* $\mathcal{SPL}$ guarantees that the structure of the composed $\mathcal{SPL}$ is correct, and each artifact has at least one correlation.

Managing variability and allowing the automated derivation of the migrated PVs are important steps towards reuse. However, investing in the $\mathcal{SPL}$ products artifacts in order to derive new products that are not provided by the $\mathcal{SPL}$ is essential. Customers are about to request new requirements on a frequent basis, thus, new products are supposed to be derived on a continuous basis. In the next chapter, we address this need, and present our approach in supporting software engineers in deriving new software products.

CHAPTER 5

CONFIGURATION PROCESS

Contents

5.1 Introduction **74**
5.2 Configuration **75**
5.2.1 Free Feature Model Generation 78
5.3 Configuration Modes **78**
5.3.1 Restrictive Mode 79
5.3.2 Free Mode 79
5.4 Configuration Scenarios **79**
5.4.1 Possible Configuration Scenarios 79
5.5 Derivation Operations **83**
5.6 Summary **87**

5.1 Introduction

The main goal when adopting SPLs is reuse [TH03]. Reuse is an essential need when developing a family of software products. An organization that has developed a family of software products aims to target new customers that belong to the same market segment, in order to increase its profits. Thus, reusing the ready-made software products to respond to the needs of the new customers increases the orgranization profits. By enabling reuse, SPLs decrease development costs and time to market and increase software quality [PBV05]. Moreover, an organization must deal with the continuous market demands and technology changes that involve the development of new software products [HVLG12]. Since the new products respond to the same market segment, they are supposed to implement some features that are already implemented by the existing products. Therefore, instead of redeveloping the assets corresponding to those features, software engineers seek to reuse the existing products artifacts and develop only the assets required to implement the new features. For this purpose, the products implementing the required features have to be determined and mappings between features and the assets contributing in their implementation have to be identified. In the previous chapter, we proposed in our approach a mechanism to identify those mappings, and we called them *correlations*.

Prior to the derivation of a product variant, the SPL feature model has to be configured, in order to select the features required for the derivation. Further, some new features that are not offered by the existing PVs might be required. Hence, in this chapter, we address **Challenge 1.b** of this dissertation, by presenting how we support the configuration process, that is a preliminary step before deriving a product variant. As well, we address **Challenge 2.a** in terms of allowing the addition of new features during the configuration. Our approach supports the derivation of new products and allows the derivation of existing ones as well. We identify four possible situations that can arise. A software engineer might proceed with a valid configuration, by selecting a set of features that are implemented exactly by an existing product. Hence, an automated derivation of the corresponding product has to be done. Figure 5.1a illustrates this situation. A second situation occurs when the configuration is valid, but still new features have to be added. Hence, in our approach, we enable the addition of new features directly during the configuration. Similarly to the previous situation, the corresponding product is provided, however, a manual completion is required to add the new features and thus, create a new product. Figure 5.1b illustrates this situation. Another situation might occur, in which the software engineer selects a set of features that are part of the implementation of an existing product or spread on several products. Hence, the configuration is invalid, since no existing product implements all and only the required features. Therefore, a new product has to be constructed. In this situation, we aim to support the derivation of the possible scenarios and operations to perform in order to derive the new product. We call the product to derive the "desired product". Figure 5.1c illustrates the third situation. Finally, a software engineer might select features corresponding to an invalid configuration, but also adds new features. Same as previous situation, we aim to support the derivation with the possible scenarios and operations and let software engineer add the new features. Figure 5.1d illustrates the fourth situation. To enable invalid configurations, we generate a *free FM* that corresponds to a constraint-free version of the *restrictive FM*. Therefore, the selection of any possible combination of features during a configuration is now possible (**Objective 17**). Further, we suggest the possible *scenarios* by means of *operations* to perform at asset level, that software engineers can rely on to derive the desired product (**Objective 12**). Several scenarios might be possible to derive a certain product, and several operations might be possible at each asset level. Therefore, instead of proposing a single solution, we propose to

software engineers the possible scenarios and operations. Hence, we guide the derivation without automating it or imposing a specific solution (**Objective 14**). Thus, we keep decision-making for software engineers that construct the products on their own.

5.2 Configuration

When a software engineer decides to derive a product, she has to identify the required features, in order to figure out if there exists a product that implements entirely those features. This process is done systematically when configuring the FM of an SPL, that we call in our approach *restrictive FM*. However, the restrictive FM limits the configuration to the exact set of products offered by the $\mathcal{SPL}$. We highlighted earlier on the importance of developing new software products, in order to satisfy the arising market demands. A new product might require the implementation of new features that were not offered by the $\mathcal{SPL}$ products. Thus, in our approach, we are interested in supporting organizations in growing their product line by developing new products. Therefore, we enable the integration of new features in the $\mathcal{SPL}$, by defining them through the configuration of a new product.

Definition 5.2.1 (Configuration) :
We define a configuration cf as a triple $\langle id, \{f_j, ..., f_n\}, \{f_x, ..., f_z\}\rangle$ where id is the identifier of the configuration. The set of features $\{f_j, ..., f_n\}$ are the features required to build the desired product and offered by the $\mathcal{SPL}$, while $\{f_x, ..., f_z\}$ are the new features required to build the desired product and not offered by the $\mathcal{SPL}$. We note $\{f_j, ..., f_n\}$ as $EF(cf)$, where $EF(cf) \subseteq \mathcal{F}$ stands for *existing features* and $\{f_x, ..., f_z\}$ noted as $NF(cf) \not\subseteq \mathcal{F}$ stands for *new features*.

During the configuration of an FM, we identify three categories of features:

— *Required existing feature*: a selected feature implemented by one or more products of the $\mathcal{SPL}$ and required for the derivation of the desired product.

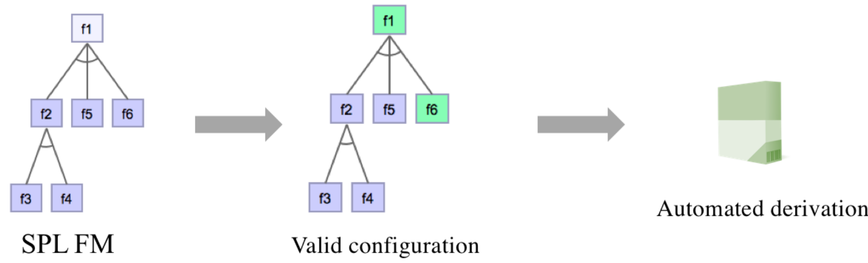
— *Unrequired existing feature*: an unselected feature implemented by one or more products of the $\mathcal{SPL}$ and not required for the derivation of the desired product.

— *Required new feature*: a selected feature that is not implemented by any of the $\mathcal{SPL}$ products, but required for the derivation of the desired product.

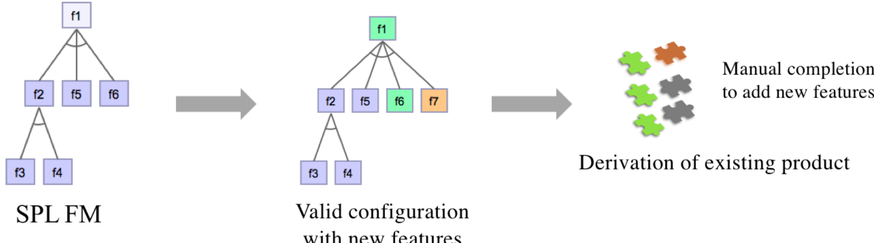
Inspired from [KNGDNK+16], we determine the following properties, that specify the relationship between a configuration and a $\mathcal{SPL}$ product:

Property 5.2.1 (Product contributes in configuration) :
An $\mathcal{SPL}$ product $p \in \mathcal{P}$ *contributes in* a configuration cf if p implements at least one feature required by cf, thus $EF(cf) \cap F(p) \neq \{\phi\}$. In other words, $\exists f_j \in F(p), f_j \in EF(cf)$.

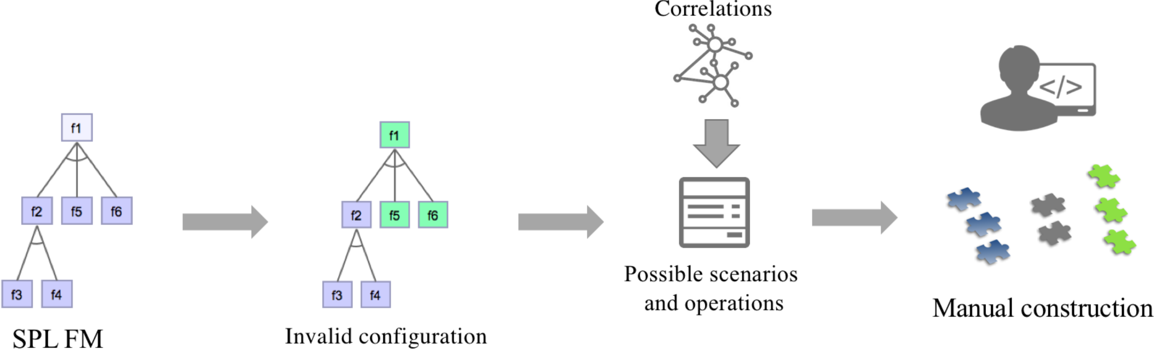
Property 5.2.2 (Product realizes a configuration) :
An $\mathcal{SPL}$ product $p \in \mathcal{P}$ *realizes* a configuration cf if all and only the features required by cf are implemented by p, thus if $EF(cf) = F(p)$. In other words, $\forall f_j \in EF(cf), f_j \in F(p) \wedge \nexists f_k \in F(p), f_k \notin EF(cf)$.



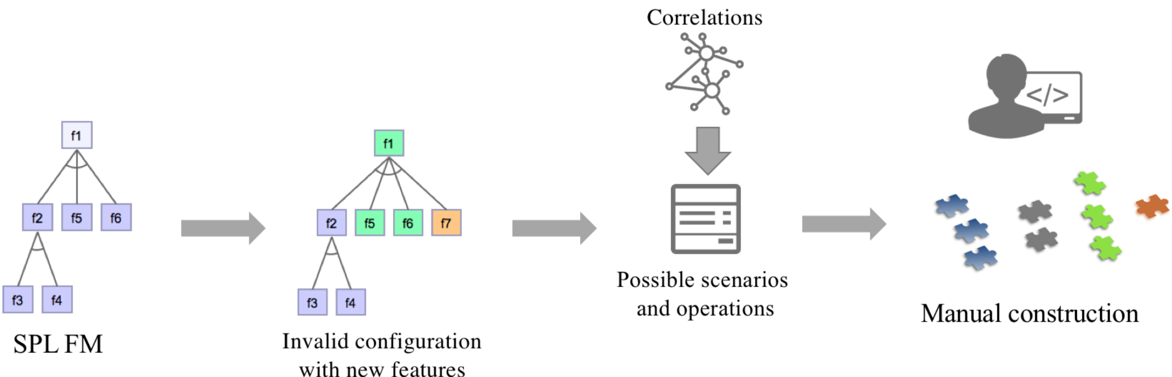
(a) Derivation of existing product



(b) Derivation of new product by adding new features to existing one



(c) Derivation of new product by using existing features



(d) Derivation of new product by using existing features and adding new ones

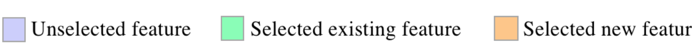


Figure 5.1: Derivation process

Property 5.2.3 (Product covers a configuration) :
An $\mathcal{SPL}$ product $p \in \mathcal{P}$ *covers* a configuration cf if all features required by cf are implemented by p, but p implements in addition some features that are not required by cf, thus if $EF(cf) \subset F(p)$. In other words, $\forall f_j \in EF(cf), f_j \in F(p) \wedge \exists f_k \in F(p), f_k \notin EF(cf)$.

Figure 5.2 shows an $\mathcal{SPL}$ example that consists of four products and illustrates the features implemented by each product. Given a configuration cf_k where $EF(cf_k) = \{f_4, f_7\}$, we can say that:

— p_1 contributes in cf_k, since $EF(cf_k) \cap F(p_1) = \{f_4\}$.

— p_2 covers cf_k, since $EF(cf_k) \subset F(p_2)$, where $\{f_4, f_7\} \subset \{f_1, f_2, f_3, f_4, f_5, f_6, f_7\}$.

— p_3 covers cf_k, since $EF(cf_k) \subset F(p_3)$, where $\{f_4, f_7\} \subset \{f_4, f_6, f_7\}$.

— p_4 realizes cf_k, since $EF(cf_k) = F(p_4) = \{f_4, f_7\}$.

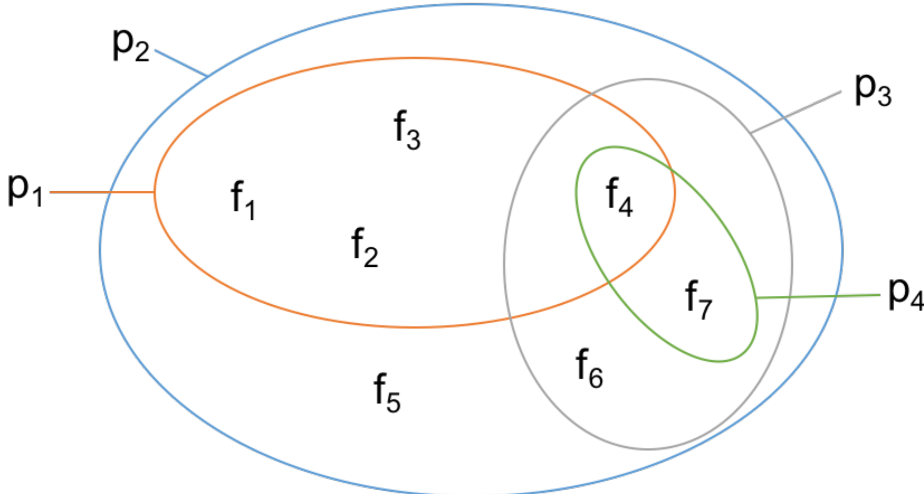


Figure 5.2: An $\mathcal{SPL}$ example

Several questions might arise during the configuration of the restrictive FM, in order to derive the desired product:

1. Is there a product in $\mathcal{SPL}$ that *realizes* the configuration, or do we have to break the constraints imposed by the restrictive FM in order to achieve the configuration?

2. Does the $\mathcal{SPL}$ offers all the features required by the desired product? Or there exists some new required features that are not offered by the $\mathcal{SPL}$ products?

3. If new features are required, how can a software engineer express the need of new features during the configuration and how those features can be added to the $\mathcal{SPL}$ FM?

Selecting a set of features that do not correspond to a valid configuration is not possible via the *restrictive FM*. A configuration via the *restrictive FM* is restricted by the constraints that define the variability between its features. Therefore, the only possible configurations that can be made through restrictive FM are the ones for which there exists a product that realizes

the configuration. On the other side, in order to introduce new features required by the desired product, we enable the addition of new features through a configuration made on a restrictive FM. Thus, despite that the *restrictive FM* allows an automated derivation of the $\mathcal{SPL}$ products, it does not permit the configuration of new ones, except the ones that involve the addition of new features on top of a product that realizes the configuration.

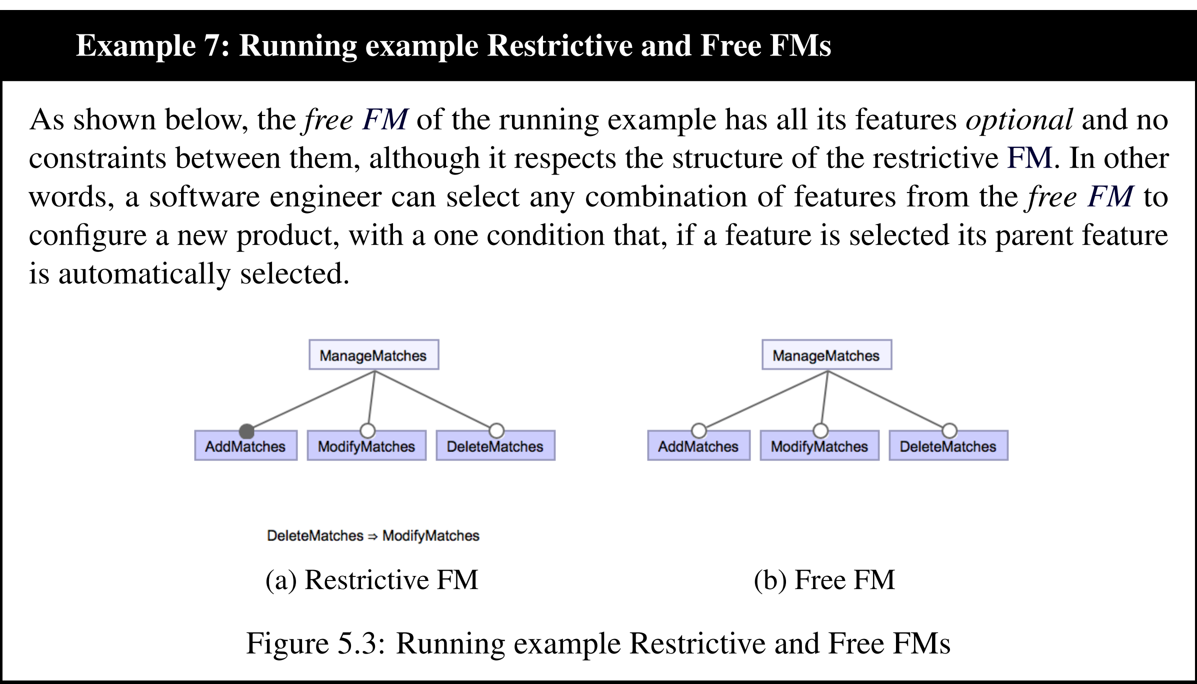
5.2.1 Free Feature Model Generation

Since a restrictive FM does not allow to break the constraints imposed by the $\mathcal{SPL}$ variability, we define a *free* FM in which a software engineer can select a set of features that does not correspond to a valid configuration.

Definition 5.2.2 (Free FM) :
A *free FM* is a constraint-free feature model that includes all the features of a *restrictive FM* where all features are optional except the root feature, without defining any constraint.

Example 7: Running example Restrictive and Free FMs

As shown below, the *free FM* of the running example has all its features *optional* and no constraints between them, although it respects the structure of the restrictive FM. In other words, a software engineer can select any combination of features from the *free FM* to configure a new product, with a one condition that, if a feature is selected its parent feature is automatically selected.



(a) Restrictive FM (b) Free FM

Figure 5.3: Running example Restrictive and Free FMs

5.3 Configuration Modes

We defined for an $\mathcal{SPL}$ two FMs: *restrictive FM* and *free FM*. In derivation process, we define respectively two configuration modes – *restrictive* and *free* – where each employs its corresponding FM, to allow a software engineer to perform a configuration, in order to derive a product. Regardless the FM in use to configure a product, we enable the definition of new features at the configuration level, so that, software engineers can add new features to create new products, whenever those features are not provided by the $\mathcal{SPL}$.

5.3.1 Restrictive Mode

A *restrictive mode* employs the *restrictive FM*, which restricts a configuration to a set of features that correspond to an existing product. Therefore, it allows the automated derivation of existing products. Moreover, since we enabled the definition of new features at the configuration level, software engineers are able to define features in restrictive mode, and therefore, derive new products that their *required existing features* are *realized* by a product in the $\mathcal{SPL}$.

5.3.2 Free Mode

A *free mode* employs the *free FM* which is a constraint-free FM, that allows the selection of any set of features, since all features are optional. Therefore, free mode allows the derivation of a new product by collecting features from one or several existing products, in addition to the possibility of adding new features that are not provided by the $\mathcal{SPL}$.

5.4 Configuration Scenarios

When a software engineer selects a set of features in *restrictive mode*, either the configuration is *valid*, and thus, the set of *required existing features* is *realized* by one and only one product, or the configuration is *invalid*, and thus, the configuration is to be performed in *free mode*. If the set of *required existing features* is not *realized* by a single product, then, there might exist a product that covers this configuration, or there might exist a combination of products, where each product contributes in the configuration and the combination by itself realizes or covers the configuration. Therefore, we call a *configuration scenario*, a possible scenario to achieve a configuration. Indeed, a configuration can be achieved by several configuration scenarios.

Definition 5.4.1 (Configuration scenario) :
We define for each configuration cf_j a set of configuration scenarios $\{cs_1(cf_j), ..., cs_n(cf_j)\}$ noted as $CS(cf_j)$, where a configuration scenario $cs_i(cf_j)$ is defined as a pair $\langle\{\langle p_k, \{f_q, ..., f_s\}\rangle\}$, $\{f_x, ..., f_z\}\rangle$, where $\{\langle p_k, \{f_q, ..., f_s\}\rangle\}$ is a combination of products that can be reused to achieve the configuration and $\{f_x, ..., f_z\}$ is $NF(cf)$ if any. A product is candidate for a configuration scenario if it implements at least one of the features of $EF(cf)$. Further, for each combination, the unrequired features $\{f_q, ..., f_s\}$ implemented by a candidate product p_k are identified.

Property 5.4.1 (Products of a configuration scenario) :
Given a configuration scenario $cs_i(cf_j)$ of a configuration cf_j, the combination of products participating in $cs_i(cf_j)$ are noted as $P(cs_i(cf_j))$.

5.4.1 Possible Configuration Scenarios

Since several scenarios might be determined for a certain configuration, we identify the possible configuration scenarios to achieve a configuration as follows:

Integrality: A configuration cf_j can be achieved by *integrality (S1 in Figure 5.4)* if there exists a configuration scenario $cs_i(cf_j)$ that consists of a product in the $\mathcal{SPL}$ that *realizes* the

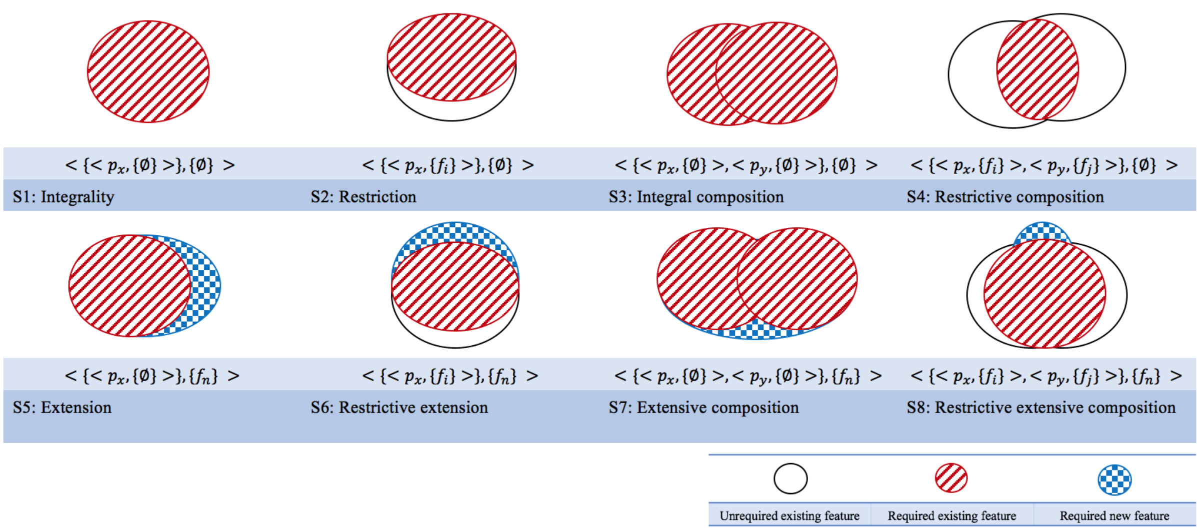


Figure 5.4: Possible scenarios to achieve a configuration

set of *required existing features* selected in the configuration. *Integrality* can arise in *restrictive* mode.

Restriction: A configuration cf_j can be achieved by *restriction (S2 in Figure 5.4)* if there exists a configuration scenario $cs_i(cf_j)$ that consists of a product in the $\mathcal{SPL}$ that *covers* the set of *required existing features* selected in the configuration. *Restriction* can arise in *free* mode.

Integral composition: A configuration cf_j can be achieved by *integral composition (S3 in Figure 5.4)* if there exists a configuration scenario $cs_i(cf_j)$ that consists of a combination of products in the $\mathcal{SPL}$ where each product *contributes in* the configuration, and the combination *realizes* the set of *required existing features* selected in the configuration. *Integral composition* can arise in *free* mode.

Restrictive composition: A configuration cf_j can be achieved by *restrictive composition (S4 in Figure 5.4)* if there exists a configuration scenario $cs_i(cf_j)$ that consists of a combination of products in the $\mathcal{SPL}$ where each product *contributes in* or *covers* the configuration, and the combination *covers* the set of *required existing features* selected in the configuration. *Restrictive composition* can arise in *free* mode.

Extension: A configuration cf_j can be achieved by *extension (S5 in Figure 5.4)* if there exists a configuration scenario $cs_i(cf_j)$ that consists of a product in the $\mathcal{SPL}$ that *realizes* the set of *required existing features* selected in the configuration, and the configuration has in addition some selected *required new features*. *Extension* can arise in *restrictive* mode. An *extension* is an *integrality* with *required new features*.

Restrictive extension: A configuration cf_j can be achieved by *restrictive extension (S6 in Figure 5.4)* if there exists a configuration scenario $cs_i(cf_j)$ that consists of a product in the $\mathcal{SPL}$ that *covers* the set of *required existing features* selected in the configuration, and the configuration has in addition some selected *required new features*. *Restrictive extension* can arise in *restrictive* mode. A *restrictive extension* is a *restriction* with *required new features*.

Extensive composition: A configuration cf_j can be achieved by *extensive composition (S7 in Figure 5.4)* if there exists a configuration scenario $cs_i(cf_j)$ that consists of a combination of products in the $\mathcal{SPL}$ where each product *contributes in* the configuration, and the combination *realizes* the set of *required existing features* selected in the configuration, and the configuration has in addition some selected *required new features*. *Extensive composition* can arise in *free* mode. An *extensive composition* is an *integral composition* with *required new features*.

Restrictive extensive composition: A configuration cf_j can be achieved by *restrictive extensive composition (S8 in Figure 5.4)* if there exists a configuration scenario $cs_i(cf_j)$ that consists of a combination of products in the $\mathcal{SPL}$ where each product *contributes in* or *covers* the configuration, and the combination *covers* the set of *required existing features* selected in the configuration, and the configuration has in addition some selected *required new features*. *Restrictive extensive composition* can arise in *free* mode. A *restrictive extensive composition* is a *restrictive composition* with *required new features*.

Given the $\mathcal{SPL}$ shown in Figure 5.2 that consists of four products, we demonstrate on some configurations made in both restrictive and free mode, the possible configuration scenarios that we presented above.

Restrictive mode configuration scenarios:

Let cf_5 a new configuration, where $EF(cf_5) = \{f_1, f_2, f_3, f_4\}$ and $NF(cf_5) = \{\phi\}$. The product p_1 *realizes* cf_5 since $F(p_1) = EF(cf_5)$. Thus, $cs_1(cf_5) = \langle\{\langle p_1, \{\phi\}\rangle\}, \{\phi\}\rangle$ *(integrality).*

Let cf_6 a new configuration, where $EF(cf_6) = \{f_4, f_7\}$ and $NF(cf_6) = \{f_8\}$. The product p_4 *realizes* cf_6 since $F(p_4) = EF(cf_6)$. Thus, $cs_1(cf_6) = \langle\{\langle p_4, \{\phi\}\rangle\}, \{f_8\}\rangle$ *(extension).*

Whenever exists a product p_j that *realizes* a configuration cf_k where $F(p_j) = EF(cf_k)$, the only configuration scenario to be considered for this configuration is the $integrality$ that is achieved by p_j if $NF(cf_k) = \{\phi\}$ or the *extension* if $NF(cf_k) \neq \{\phi\}$.

Free mode configuration scenarios

Let cf_7 a new configuration, where $EF(cf_7) = \{f_1, f_2, f_3, f_4, f_7\}$ and $NF(cf_7) = \{\phi\}$. No product in $\mathcal{SPL}$ *realizes* cf_7. However, several configuration scenarios are possible for cf_7:

— $cs_1(cf_7) = \langle\{\langle p_2, \{f_5, f_6\}\rangle\}, \{\phi\}\rangle$ *(restriction).*

— $cs_2(cf_7) = \langle\{\langle p_1, \{\phi\}\rangle, \langle p_2, \{f_5, f_6\}\rangle\}, \{\phi\}\rangle$ *(restrictive composition).*

— $cs_3(cf_7) = \langle\{\langle p_1, \{\phi\}\rangle, \langle p_3, \{f_6\}\rangle\}, \{\phi\}\rangle$ *(restrictive composition).*

— $cs_4(cf_7) = \langle\{\langle p_1, \{\phi\}\rangle, \langle p_4, \{\phi\}\rangle\}, \{\phi\}\rangle$ *(integral composition).*

— $cs_5(cf_7) = \langle\{\langle p_2, \{f_5, f_6\}\rangle, \langle p_3, \{f_6\}\rangle\}, \{\phi\}\rangle$ *(restrictive composition).*

— $cs_6(cf_7) = \langle\{\langle p_2, \{f_5, f_6\}\rangle, \langle p_4, \{\phi\}\rangle\}, \{\phi\}\rangle$ *(restrictive composition).*

— $cs_7(cf_7) = \langle\{\langle p_1, \{\phi\}\rangle, \langle p_2, \{f_5, f_6\}\rangle, \langle p_3, \{f_6\}\rangle\}, \{\phi\}\rangle$ *(restrictive composition).*

— $cs_8(cf_7) = \langle\{\langle p_1, \{\phi\}\rangle, \langle p_2, \{f_5, f_6\}\rangle, \langle p_4, \{\phi\}\rangle\}, \{\phi\}\rangle$ *(restrictive composition).*

— $cs_9(cf_7) = \langle\{\langle p_2, \{f_5, f_6\}\rangle, \langle p_3, \{f_6\}\rangle, \langle p_4, \{\phi\}\rangle\}, \{\phi\}\rangle$ *(restrictive composition).*

— $cs_{10}(cf_7) = \langle\{\langle p_1, \{\phi\}\rangle, \langle p_2, \{f_5, f_6\}\rangle, \langle p_3, \{f_6\}\rangle, \langle p_4, \{\phi\}\rangle\}, \{\phi\}\rangle$ *(restrictive composition).*

Let cf_8 a new configuration, where $EF(cf_8) = \{f_1, f_2, f_3\}$ and $NF(cf_8) = \{f_9, f_{10}\}$. No product in $\mathcal{SPL}$ *realizes* cf_8. However, several configuration scenarios are possible for cf_8:

— $cs_1(cf_8) = \langle\{\langle p_1, \{f_4\}\rangle\}, \{f_9, f_{10}\}\rangle$ *(restrictive extension).*

— $cs_2(cf_8) = \langle\{\langle p_2, \{f_5, f_6, f_7\}\rangle\}, \{f_9, f_{10}\}\rangle$ *(restrictive extension).*

— $cs_3(cf_8) = \langle\{\langle p_1, \{f_4\}\rangle, \langle p_2, \{f_5, f_6, f_7\}\rangle\}, \{f_9, f_{10}\}\rangle$ *(restrictive extensive composition).*

Example 8: Running example configuration of a new product

Given the products of the running example and their corresponding features as shown in Table 1.1. Let cf_4 a new configuration, where $EF(cf_4) = \{ManageMatches, AddMatches, DeleteMatches\}$ and $NF(cf_4) = \{\phi\}$. No product *realizes* cf_4, however, several configuration scenarios are possible:

— $cs_1(cf_4) = \langle\{\langle p_2, \{ModifyMatches\}\rangle\}, \{\phi\}\rangle$ *(restriction)*

— $cs_2(cf_4) = \langle\{\langle p_1, \{ModifyMatches\}\rangle, \langle p_2, \{ModifyMatches\}\rangle\}, \{\phi\}\rangle$ *(restrictive composition)*

— $cs_3(cf_4) = \langle\{\langle p_2, \{ModifyMatches\}\rangle, \langle p_3, \{\phi\}\rangle\}, \{\phi\}\rangle$ *(restrictive composition)*

— $cs_4(cf_4) = \langle\{\langle p_1, \{ModifyMatches\}\rangle, \langle p_2, \{ModifyMatches\}\rangle, \langle p_3, \{\phi\}\rangle\}, \{\phi\}\rangle$ *(restrictive composition)*

When a configuration is achieved in free mode, where no product *realizes* it, the products or combinations of products that *realize* or *cover* the *required existing features* are identified via configuration scenarios. Most likely, several configuration scenarios are found for a certain configuration. Thus, a software engineer has to chose the "appropriate" configuration scenario to derive the desired product. However, the question is: "how to determine the appropriate configuration scenario?".

One might consider a *restrictive composition* more expensive in terms of time and development efforts compared to *restriction*, since *restrictive composition* involves more products. Also, logically someone might consider a *restrictive composition* involving n products more expensive than a *restrictive composition* involving $n-1$ products. Moreover, one might consider a *restriction* more expensive than an *integral composition*, since *restriction* requires the removal of some features. On the contrary, someone else might consider *integral composition* more expensive than *restriction*, by considering that removal of some features is less expensive than integrating assets from several products to derive a new product. As demonstrated, it is ambiguous for us to determine what is the "appropriate" configuration scenario to derive a desired

product, if the choice has to be taken according to the products and features involved in the configuration scenario. However, the software developer who developed the product variants is the one responsible to determine the "appropriate" solution, since she knows better the code and how the software variants are composed. Therefore, we aim to suggest to end users the possible solutions and let them be the decision makers.

5.5 Derivation Operations

The implementation of the $\mathcal{SPL}$ products is achieved via their asset instances. Thus, to help software engineers to determine the "appropriate" configuration scenario, we must take into consideration the operations required to derive a desired product according to the asset instances exploited by the products involved in each configuration scenario. This process has to be realized through the following steps:

1. Identify the assets required to implement the desired product.
2. Determine the possible operations to perform on each asset, in order to obtain an asset instance that fulfills the implementation of the desired product features.
3. Map each configuration scenario to its corresponding operations.

These steps involve the examination of the correlations identified between the features and the assets as well as their instances.

Definition 5.5.1 (Required assets for derivation) :
An asset a is required for deriving a desired product, if there exists at least a feature f such as, f is in correlation with a and f is one of the *required existing features* of the configuration cf built to achieve the derivation of the desired product. Hence, we note the *required assets* of cf as $A(cf)$, where $A(cf) = \{a \mid \exists f \in EF(cf), c(f, a)\}$.

For each identified required asset, several operations might be possible to produce an asset instance that serves for the implementation of the required features of the configuration.

1. An asset instance has to be *cloned*,
2. then modified if necessary to *remove* implementation fragments corresponding to some features that it implements and *unrequired* by the configuration,
3. and if there still exists some *required* features that are not implemented by the *cloned instance*, their implementation fragments must be *extracted* from the other instances of the asset and integrated in the clone.

The resulting asset instance of each required asset has to implement the set of features $F(a) \bigcap EF(cf)$. Thus, we identify three types of actions that might be taken over the instances of a required asset in order to produce the desired instance:

1. Clone and Retain (CRT): clone an asset instance and retain it as it is, without modifying its implementation.

2. Clone and Remove (CRM): clone an asset instance, and remove from it the implementation fragments corresponding to the features that it is in correlation with but are not required by the configuration.

3. Extract and Add (ETA): extract from an asset instance the implementation fragments of some features required by the configuration, and add them to a cloned instance under construction. An ETA action is used only as a subsequent to a CRT or CRM action in order to complete the construction of a cloned instance with extracted implementation fragments.

Definition 5.5.2 (Action) :
An *action* ac is defined as a triple $\langle type, a^{\mathrm{i}}, \{f_j, ..., f_n\}\rangle$, where $type$ corresponds to one of the types defined above: $\{CRT, CRM, ETA\}$. For CRT and CRM actions, a^{i} corresponds to the asset instance to clone. For an ETA action, a^{i} corresponds to an asset instance to extract from. Whereas, $\{f_j, ..., f_n\}$ corresponds to the set of features to remove from a^{i} if the action is CRM, or to extract from a^{i} if the action is ETA, while it is an empty set for a CRT action.

Hence, the resulting asset instance for a required asset is produced by cloning an asset instance exploited by a product of the configuration scenario using a CRT or CRM action, removing the implementation fragments corresponding to the unrequired features in case of a CRM action, and extracting the remaining required features from other instances using an ETA action, if any.

Example 9: Some actions on a configuration from running example

Based on Example 8, $EF(cf_4) = \{ManageMatches, AddMatches, DeleteMatches\}$ and according to Table 4.3, the asset $SaveMatch.java$ is in correlation with the features $ManageMatches$, $AddMatches$ and $ModifyMatches$. Thus, the asset $SaveMatch.java$ is required and the resulting instance has to integrate implementation fragments corresponding $EF(cf_4) \bigcap F(SaveMatch.java) = \{ManageMatches, AddMatches\}$. The resulting instance might be a possible result of a CRT action, which consists of cloning the instance $SaveMatch.java^2$ that is in correlation with $\{ManageMatches, AddMatches\}$ and retaining it as it is. Another possible result can be obtained using a CRM action applied on instance $SaveMatch.java^1$ by cloning it and removing the implementation fragments corresponding to the unrequired feature $ModifyMatches$.

It is required sometimes to make several actions in order to obtain the needed asset instance. This occurs when a CRT or CRM action is accompanied with one or more ETA actions on several instances of the asset, in order to create a new instance. We call this set of actions as "operations".

Definition 5.5.3 (Operation) :
We define an *operation* as the set of actions needed to produce the desired asset instance. Thus, an operation op is a triple $\langle a, \{ac_1, ..., ac_n\}, a^{\mathrm{i}}\rangle$, where $a \in A(cf)$ is the required asset, and $\{ac_1, ..., ac_n\}$ noted as $AC(op)$ is the set of actions to be made to obtain the desired asset instance a^{i}.

We categorize the possible operations to perform into three groups:

— **Concludable operations:** an operation is considered *concludable* if it is composed of one and only one CRT action. Hence, it is not supposed to require efforts except a clone of its corresponding asset instance.

— **Substitutional operations:** an operation is considered *substitutional* if there exists an asset instance in $\mathcal{SPL}$ that is in correlation with all and only the features required for the asset instance that it is supposed to produce. Therefore, a *substitutional* operation is a kind of alert that warns a software developer that there already exists an asset instance exploited by a product that is not part of the configuration scenario, and that asset instance implements the exact set of required features. This situation occurs on the first operation corresponding to the asset $SaveMatch.java$ presented in Table 5.1.

— **Constructive operations:** an operation is considered *constructive* if it aims to deliver a new asset instance that is not provided by the $\mathcal{SPL}$. Hence, it requires some efforts from a software developer to produce the asset instance.

Example 10: Some operations on a configuration from running example

Given the asset $style.css$ of the running example, and based on configuration cf_4 introduced in Example 8, we identify the possible operations to perform at the level of the asset $style.css$, in order to obtain the desired instance:

— $op_1 = \langle style.css, \{\langle CRM, style.css^1, \{ModifyMatches\}\rangle, \langle ETA, style.css^2, \{DeleteMatches\}\rangle\}, style.css^4\rangle$ consists of cloning the asset instance $style.css^1$ and removing from it the implementation fragments corresponding to the feature $ModifyMatches$, then extracting the implementation fragments corresponding to the feature $DeleteMatches$ from $style.css^2$ and adding them to the clone, in order to obtain the desired instance $style.css^4$.

— $op_2 = \langle style.css, \{\langle CRM, style.css^2, \{ModifyMatches\}\rangle\}, style.css^4\rangle$ consists of cloning the asset instance $style.css^2$ and removing from it the implementation fragments corresponding to the feature $ModifyMatches$ in order to obtain the desired instance $style.css^4$.

— $op_3 = \langle style.css, \{\langle CRT, style.css^3, \{\phi\}\rangle, \langle ETA, style.css^2, \{DeleteMatches\}\rangle\}, style.css^4\rangle$ consists of cloning the asset instance $style.css^3$, then extracting the implementation fragments corresponding to the feature $DeleteMatches$ from $style.css^2$ and adding them to the clone, in order to obtain the desired instance $style.css^4$.

Table 5.1 shows the possible configuration scenarios corresponding to the configuration cf_4 and their possible operations to perform.

Once operations are identified, they are mapped to the configuration scenarios. For each configuration scenario, several operations are assigned. Precisely, at the level of each required asset, one or several operations might be possible to construct the desired instance of the asset. Therefore, to achieve the derivation of a new product variant, a software developer is provided with all possible configuration scenarios, and all possible operations that can be performed at asset level. For each required asset, only one operation has to be selected.

Table 5.1: Possible configuration scenarios of cf_4 with their possible operations

Configuration scenario	
$cs_1(cf_4) = \langle\{\langle p_2, \{ModifyMatches\}\rangle\}, \{\phi\}\rangle$	
Asset	**Operations**
$match.jsp$	$\langle match.jsp, \{\langle CRT, match.jsp^1, \{\phi\}\rangle\}, match.jsp^1\rangle$
$SaveMatch.java$	$\langle SaveMatch.java, \{\langle CRM, SaveMatch.java^1, \{ModifyMatches\}\rangle\}, SaveMatch.java^2\rangle$
$style.css$	$\langle style.css, \{\langle CRM, style.css^2, \{ModifyMatches\}\rangle\}, style.css^4\rangle$
$DeleteMatch.java$	$\langle DeleteMatch.java, \{\langle CRT, DeleteMatch.java^1, \{\phi\}\rangle\}, DeleteMatch.java^1\rangle$
Configuration scenario	
$cs_2(cf_4) = \langle\{\langle p_1, \{ModifyMatches\}\rangle, \langle p_2, \{ModifyMatches\}\rangle\}, \{\phi\}\rangle$	
Asset	**Operations**
$match.jsp$	$\langle match.jsp, \{\langle CRT, match.jsp^1, \{\phi\}\rangle\}, match.jsp^1\rangle$
$SaveMatch.java$	$\langle SaveMatch.java, \{\langle CRM, SaveMatch.java^1, \{ModifyMatches\}\rangle\}, SaveMatch.java^2\rangle$
$style.css$	$\langle style.css, \{\langle CRM, style.css^1, \{ModifyMatches\}\rangle, \langle ETA, style.css^2, \{DeleteMatches\}\rangle\}, style.css^4\rangle$
	$\langle style.css, \{\langle CRM, style.css^2, \{ModifyMatches\}\rangle\}, style.css^4\rangle$
$DeleteMatch.java$	$\langle DeleteMatch.java, \{\langle CRT, DeleteMatch.java^1, \{\phi\}\rangle\}, DeleteMatch.java^1\rangle$
Configuration scenario	
$cs_3(cf_4) = \langle\{\langle p_2, \{ModifyMatches\}\rangle, \langle p_3, \{\phi\}\rangle\}, \{\phi\}\rangle$	
Asset	**Operations**
$match.jsp$	$\langle match.jsp, \{\langle CRT, match.jsp^1, \{\phi\}\rangle\}, match.jsp^1\rangle$
$SaveMatch.java$	$\langle SaveMatch.java, \{\langle CRM, SaveMatch.java^1, \{ModifyMatches\}\rangle\}, SaveMatch.java^2\rangle$
	$\langle SaveMatch.java, \{\langle CRT, SaveMatch.java^2, \{\phi\}\rangle\}, SaveMatch.java^2\rangle$
$style.css$	$\langle style.css, \{\langle CRM, style.css^2, \{ModifyMatches\}\rangle\}, style.css^4\rangle$
	$\langle style.css, \{\langle CRT, style.css^3, \{\phi\}\rangle, \langle ETA, style.css^2, \{DeleteMatches\}\rangle\}, style.css^4\rangle$
$DeleteMatch.java$	$\langle DeleteMatch.java, \{\langle CRT, DeleteMatch.java^1, \{\phi\}\rangle\}, DeleteMatch.java^1\rangle$
Configuration scenario	
$cs_4(cf_4) = \langle\{\langle p_1, \{ModifyMatches\}\rangle, \langle p_2, \{ModifyMatches\}\rangle, \langle p_3, \{\phi\}\rangle\}, \{\phi\}\rangle$	
Asset	**Operations**
$match.jsp$	$\langle match.jsp, \{\langle CRT, match.jsp^1, \{\phi\}\rangle\}, match.jsp^1\rangle$
$SaveMatch.java$	$\langle SaveMatch.java, \{\langle CRM, SaveMatch.java^1, \{ModifyMatches\}\rangle\}, SaveMatch.java^2\rangle$
	$\langle SaveMatch.java, \{\langle CRT, SaveMatch.java^2, \{\phi\}\rangle\}, SaveMatch.java^2\rangle$
$style.css$	$\langle style.css, \{\langle CRM, style.css^1, \{ModifyMatches\}\rangle, \langle ETA, style.css^2, \{DeleteMatches\}\rangle\}, style.css^4\rangle$
	$\langle style.css, \{\langle CRM, style.css^2, \{ModifyMatches\}\rangle\}, style.css^4\rangle$
	$\langle style.css, \{\langle CRT, style.css^3, \{\phi\}\rangle, \langle ETA, style.css^2, \{DeleteMatches\}\rangle\}, style.css^4\rangle$
$DeleteMatch.java$	$\langle DeleteMatch.java, \{\langle CRT, DeleteMatch.java^1, \{\phi\}\rangle\}, DeleteMatch.java^1\rangle$

5.6 Summary

In this chapter, we presented how we support the configuration of a product variant prior to its derivation in two configuration modes. A configuration made in restrictive mode allows an automated derivation of an existing product or the derivation of a new product based on the integration of new features into an existing one. A configuration made in free mode allows the derivation of a new product. We guide this derivation by providing the possible configuration scenarios and the operations to perform at each required asset level in order to construct the asset instances of the desired product.

Since several configuration scenarios are possible and several operations might be identified at an asset level, we keep the choice of the appropriate configuration scenario and operations to the software developer who is supposed to be familiar with product variants and their source code. In this upcoming chapter, we introduce an additional parameter to support the product derivation by estimating the cost of the operations to perform and globally the cost of each configuration scenario. In chapter 7, we explain how we suggest to end users the possible configuration scenarios and operations, in a way that facilitates on them the selection of the appropriate ones according to their own preferences.

CHAPTER 6

TOWARDS COST-ESTIMATED DERIVATION

Contents

6.1 Introduction **90**
6.2 Cost-Estimation **90**
6.2.1 Action Type Weight 90
6.2.2 Correlation Degree 90
6.2.3 Action Cost 92
6.2.4 Operation Cost 94
6.2.5 Configuration Scenario Cost 95
6.3 Summary **96**

6.1 Introduction

In the previous chapter, we proposed two configuration modes to support the configuration prior to the derivation of a product variant. We presented how we provide the possible configuration scenarios and operations to perform to fulfill the derivation. We highlighted earlier on the importance of assigning the decision to make to the software engineers who are supposed to be familiar with the $\mathcal{SPL}$ product and their source code, so they can choose the appropriate configuration scenario and operations for derivation. Despite that, a configuration might have a large number of possible configuration scenarios on the one side and each required asset might have a large number of possible operations on the other side. As presented in **Challenge 1.c**, this multiplicity of choices can turn the software engineers decisions difficult. For this purpose, we supply our approach with an estimation of the cost of the operations to perform and the cost of each configuration scenario, that we introduce in this chapter (**Objective 13**). This cost estimation is supposed to be an additional argument that supports software engineers in selecting a configuration scenario and the operations to perform at asset level to achieve the derivation of the desired product.

In this chapter, we demonstrate the cost estimation only on the excerpt of assets of the running example to keep the explanation simple. We note that the product variants of the running example employ more assets than the ones presented in Table 1.2, and applying the cost estimation functions on the excerpt of assets produces different cost values, however, it gives analogical results to the whole running example.

6.2 Cost-Estimation

6.2.1 Action Type Weight

The cost of an operation is estimated based on the actions that it is composed of. In case of a CRT action, an asset instance has to be cloned without being modified, therefore, no cost has to be allocated. In case of a CRM action, an asset instance has to be cloned and implementation fragments corresponding to one or more features must be removed from the cloned instance. Similarly, for an ETA action, implementation fragments corresponding to one or more features must be extracted from an instance and integrated in the cloned instance. Therefore, for both CRM and ETA a cost has to be allocated. We assume that an ETA action costs 50% additional efforts compared to a CRM action, since it consists in adding the extracted fragments to the clone after their extraction.

Definition 6.2.1 (Action type weight) :
We define an *action type weight* denoted aw, where $aw = 0$ for CRT, $aw = 1$ for CRM and $aw = 1.5$ for ETA.

6.2.2 Correlation Degree

We estimate the cost of removing or extracting a feature based on the following global assumption:

"as much as the correlation degree between a feature and an asset instance is high, the removal or extraction of the feature from the asset instance becomes hard".

We define *correlation degree* based on the following assumptions:

— *As much as the number of features that an asset instance is in correlation with ($F(a^i)$) increases, the correlation degree between the asset instance and any of those features decreases*. Hence, the impact of the feature $f \in F(a^i)$ is proportional to $1 \div |F(a^i)|$.

— *As much as the number of assets that a feature is in correlation with ($A(f)$) increases, the correlation degree between the feature and any of those assets decreases*. Hence, the impact of an asset $a \in A(f)$ is proportional to $1 \div |A(f)|$.

— *As much as the number of instances of an asset that a feature is in correlation with increases, in relation to the overall number of instances of the asset ($AI(f/a)$), the correlation degree between the feature and the asset increases*. Hence, the number of instances of a that f is in correlation with, in relation to the overall number of instances of a corresponds to $|AI(f/a)| \div |AI(a)|$.

Definition 6.2.2 (Correlation degree) :
We define a *correlation degree* between a feature f and an asset instance a^i as:

$$cd(f, a^i) = \frac{1}{|F(a^i)|} \times \frac{1}{|A(f)|} \times \frac{|AI(f/a)|}{|AI(a)|}$$

Example 11: Asset instance correlated features from running example

Referring to Table 4.4, we compute the number of features that each asset instance is in correlation with: $|F(a^i)|$.

Table 6.1: Running example asset instance correlated features

Asset$^{\textbf{Instance}}$	$\lvert F(a^{\mathbf{i}})\rvert$
$match.jsp^1$	2
$SaveMatch.java^1$	3
$SaveMatch.java^2$	2
$style.css^1$	3
$style.css^2$	4
$style.css^3$	2
$DeleteMatch.java^1$	1

6.2.3 Action Cost

Definition 6.2.3 (Action cost) :
The cost of an action is defined as the sum of the correlation degrees of the features that have to be removed or extracted from the asset instance of this action, multiplied by the *action type weight* aw.

$$Let\, ac = \langle type, a^{\text{i}},\ \{f_j, ..., f_n\}\rangle, cost(ac) = \sum_{f_j}^{f_n}(cd(f_j, a^{\text{i}}) \times aw)$$

Example 12: Feature correlated assets from running example

Referring to Table 4.3, we compute the number of assets that each feature is in correlation with: $|A(f)|$.

Table 6.2: Running example feature correlated assets

Feature	$\vert A(f)\vert$
$ManageMatches$	3
$AddMatches$	3
$ModifyMatches$	2
$DeleteMatches$	2

Example 13: Instances of assets from running example

Referring to Table 1.2, we compute the number of instances of each asset of the running example: $|AI(a)|$.

Table 6.3: Running example instances of assets

Asset	$\vert AI(a)\vert$
$match.jsp$	1
$SaveMatch.java$	2
$style.css$	3
$DeleteMatch.java$	1

Example 14: Feature correlated asset instances from running example

Referring to Table 4.4, we compute the number of instances of each asset correlated with each feature of the running example: $|AI(f/a)|$.

Table 6.4: Running example features correlated asset instances

Feature	**Asset**	$\|AI(f/a)\|$
$ManageMatches$	$match.jsp$	1
$AddMatches$	$match.jsp$	1
$ManageMatches$	$SaveMatch.java$	2
$AddMatches$	$SaveMatch.java$	2
$ModifyMatches$	$SaveMatch.java$	1
$ManageMatches$	$style.css$	3
$AddMatches$	$style.css$	3
$ModifyMatches$	$style.css$	2
$DeleteMatches$	$style.css$	1
$DeleteMatches$	$DeleteMatch.java$	1

Example 15: Correlation degrees from running example

Referring to Table 4.4å and the examples listed above, we compute the correlation degrees between the running example features and asset instances.

Table 6.5: Correlation degrees between features and asset instances of the running example

Feature	**Asset**$^{\text{instance}}$	$cd(f, a^{i})$
ManageMatches	match.jsp^{1}	0.1666
AddMatches	match.jsp^{1}	0.1666
ManageMatches	SaveMatch.java^{1}	0.1111
AddMatches	SaveMatch.java^{1}	0.1111
ModifyMatches	SaveMatch.java^{1}	0.0833
ManageMatches	SaveMatch.java^{2}	0.1666
AddMatches	SaveMatch.java^{2}	0.1666
ManageMatches	style.css^{1}	0.1111
AddMatches	style.css^{1}	0.1111
ModifyMatches	style.css^{1}	0.1111
ManageMatches	style.css^{2}	0.0833
AddMatches	style.css^{2}	0.0833
ModifyMatches	style.css^{2}	0.8333
DeleteMatches	style.css^{2}	0.0416
ManageMatches	style.css^{3}	0.1666
AddMatches	style.css^{3}	0.1666
DeleteMatches	DeleteMatch.java^{1}	0.5000

Example 16: Actions cost from running example

Referring to Table 6.5, we compute the estimated cost of the the running example actions of the configuration cf_4.

Table 6.6: Configuration cf_4 actions estimated cost

Action	$cost(ac)$
$\langle CRT, match.jsp^1, \{\phi\}\rangle$	0.0000
$\langle CRM, SaveMatch.java^1, \{ModifyMatches\}\rangle$	0.0833
$\langle CRT, SaveMatch.java^2, \{\phi\}\rangle$	0.0000
$\langle CRM, style.css^1, \{ModifyMatches\}\rangle$	0.1111
$\langle ETA, style.css^2, \{DeleteMatches\}\rangle$	0.0625
$\langle CRM, style.css^2, \{ModifyMatches\}\rangle$	0.0833
$\langle CRT, style.css^3, \{\phi\}\rangle$	0.0000
$\langle CRT, DeleteMatch.java^1, \{\phi\}\rangle$	0.0000

6.2.4 Operation Cost

Definition 6.2.4 (Operation cost) :
The cost of an operation is defined as the sum of the cost of all its actions.

$$Let\ op = \langle a, \{ac_1, ..., ac_n\}, a^{\mathrm{i}}\rangle,\ cost(op) = \sum_{j=1}^{n} cost(ac_j)$$

Example 17: Operations cost from running example

Referring to the operations of the configuration cf_4 of the running example listed in Table 5.1 and the examples listed above, we compute the estimated cost of the the running example operations of the configuration cf_4.

Table 6.7: Configuration cf_4 operations estimated cost

Operation	$cost(op)$
$\langle match.jsp, \{\langle CRT, match.jsp^1, \{\phi\}\rangle\}, match.jsp^1\rangle$	0.0000
$\langle SaveMatch.java, \{\langle CRM, SaveMatch.java^1, \{ModifyMatches\}\rangle\}, SaveMatch.java^2\rangle$	0.0833
$\langle SaveMatch.java, \{\langle CRT, SaveMatch.java^2, \{\phi\}\rangle\}, SaveMatch.java^2\rangle$	0.0000
$\langle style.css, \{\langle CRM, style.css^1, \{ModifyMatches\}\rangle, \langle ETA, style.css^2, \{DeleteMatches\}\rangle\}, style.css^4\rangle$	0.1736
$\langle style.css, \{\langle CRM, style.css^2, \{ModifyMatches\}\rangle\}, style.css^4\rangle$	0.0833
$\langle style.css, \{\langle CRT, style.css^3, \{\phi\}\rangle, \langle ETA, style.css^2, \{DeleteMatches\}\rangle\}, style.css^4\rangle$	0.0625
$\langle DeleteMatch.java, \{\langle CRT, DeleteMatch.java^1, \{\phi\}\rangle\}, DeleteMatch.java^1\rangle$	0.0000

6.2.5 Configuration Scenario Cost

To determine the cost of a configuration scenario, we have to take into consideration the operations available at each required asset level. Since several operations are possibly identified at each required asset level, we consider only the operation having the lowest cost for each asset when computing the cost of a configuration scenario.

Definition 6.2.5 (Configuration Scenario cost) :
The cost of a configuration scenario is defined as the sum of the estimated cost of the operations having the lowest cost at each asset level.

$$Let\, cs = \{op_1, ..., op_n\},\ cost(cs) = \sum_{j=1}^{n} cost(op_j)$$

Example 18: Configuration scenario cost from running example

Referring to Table 6.7, we compute the estimated cost of the the running example configuration scenarios of the configuration cf_4.

Table 6.8: Configuration cf_4 configuration scenarios estimated cost

Configuration scenario	$cost(cs_i(cf_4))$
$cs_1(cf_4)$	0.1666
$cs_2(cf_4)$	0.1666
$cs_3(cf_4)$	0.0625
$cs_4(cf_4)$	0.0625

It is possible to have several configuration scenarios with the same estimated cost. A situation that most likely occurs is when the products involved in a configuration scenario are subset of the ones involved in the other configuration scenario. Therefore, the operations having the lowest cost at each asset level are similar for those configuration scenarios and their overall cost is equal. Further, another situation that occasionally occurs is when two configuration scenarios have the same cost without having their involved set of products subset of each other. To facilitate the selection of a configuration scenario in such situations, we prioritize the configuration scenarios having the same cost as follows:

Given two configuration scenarios cs_1 and cs_2, where $cost(cs_1) = cost(cs_2)$. The configuration scenario cs_1 is given a selection priority over cs_2 if one the following conditions is true:

1. $P(cs_1) \subset P(cs_2)$
2. $|AM(cs_1)| < |AM(cs_2)|$

where $P(cs_i)$ is the set of products involved in a configuration scenario and $AM(cs_i)$ is the number of required assets in cs_i that require a modification. Thus, software engineers can rely

on this proposed prioritization, to select a configuration scenario when several ones have the same estimated cost.

Example 19: Configuration scenarios having same cost

Referring to Table 6.8 of the running example, $cs_1(cf_4)$ and $cs_2(cf_4)$ have the same cost, however, $cs_1(cf_4)$ is composed of one product (p_2) which is a subset of the products from which $cs_2(cf_4)$ is composed (p_1 and p_2). Similarly, $cs_3(cf_4)$ and $cs_4(cf_4)$ have the same cost, however, $cs_3(cf_4)$ is composed of two product (p_2 and p_3) which are subset of the products from which $cs_4(cf_4)$ is composed (p_1, p_2 and p_3).

6.3 Summary

In this chapter, we presented an additional argument to support software engineers when deriving a new product variant. This arguments consists of a cost estimation of the proposed configuration scenarios and their corresponding operations. Providing the estimated cost of the operations facilitates the selection of an operation when several operations are possible to construct an asset instance. Similarly, it facilitates the selection of a configuration scenario when several ones are proposed. Therefore, a software developer can rely on the estimated cost of the configuration scenarios as an additional parameter to make her decisions during product derivation.

CHAPTER 7

DERIVATION AND EVOLUTION PROCESS

Contents

7.1 Introduction **98**

7.2 Product Derivation **99**

7.2.1 Derivation FM 99

7.2.2 Selection Factors 100

7.2.2.1 Numbers and indicators 100

7.2.2.2 Developer Preferences 101

7.2.2.3 Cost Estimation 102

7.3 Product Line Evolution **102**

7.3.1 Product Variant Integration 103

7.3.2 Feature Models Re-generation 103

7.3.3 Correlations and Correlation Indicators Update 103

7.3.3.1 Correlations Update 104

7.3.3.2 Correlation Indicators Update 108

7.3.4 Product Line Re-Definition 108

7.4 Summary **108**

7.1 Introduction

We focused throughout our approach on keeping decision making during product derivation on behalf of software engineers. We are interested in guiding software engineers without imposing a solution on them to accomplish the derivation (**Objective 14**). In **Chapter 5** of this dissertation, we explained how we identify the possible configuration scenarios and operations to perform to derive a desired product. Hence, software engineers must have the complete freedom to select the solution that they consider suitable for derivation, based on their own preferences. Therefore, we aim to conserve the *ownership* feature of the *Clone-and-Own* approach, since software engineers rely on our guidance to build the desired product on their own. We strengthened our support in **Chapter 6**, by estimating the cost of the possible operations and respectively the cost of the possible configuration scenarios. Thus, to derive the desired product, a software engineer can select the appropriate configuration scenario and operations and rely on the estimated costs as an additional argument of support.

In this chapter, we address **Challenge 2.b** by explaining how product derivation can be accomplished and how the $\mathcal{SPL}$ can be evolved after the derivation, by integrating the derived product into it. Hence, we adopt a reactive SPL evolution (**Objective 15**). To evolve the $\mathcal{SPL}$, when integrating a new product variant, we perform an automated and incremental update of the $\mathcal{SPL}$ correlations. This step involves further, an update of the correlation indicators that we introduced in **Chapter 6**, in order to keep the cost estimation values consistent. Moreover, we address **Challenge 2.a** to re-structure the features after adding a new product. Hence, to enable the selection of the newly added features in the future configurations, we re-generating the $\mathcal{SPL}$ restrictive FM and free FM (**Objective 18**). The $\mathcal{SPL}$ evolution process is presented in Figure 7.1.

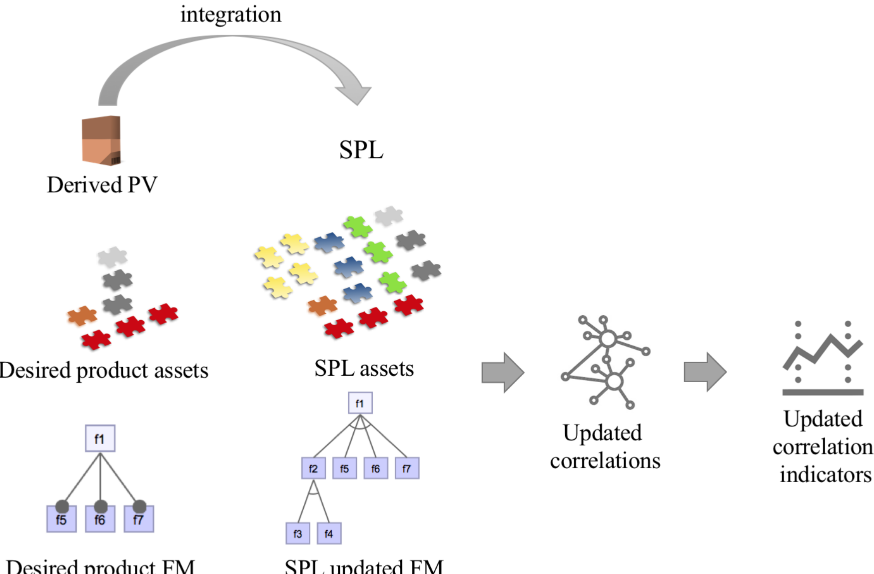


Figure 7.1: $\mathcal{SPL}$ evolution process.

7.2 Product Derivation

7.2.1 Derivation FM

Several configuration scenarios might be possible to derive a new product variant. Respectively, for each required asset, several operations might be proposed, where only one has to be chosen. We mentioned earlier that we are interested in allowing software engineers to select either a configuration scenario and operations corresponding to it, or select operations from different configuration scenarios. Furthermore, we aim to provide software engineers with additional arguments, to allow them to select a configuration scenario and operations based on their own preferences. Such arguments concern the products involved by each configuration scenario and the asset instances involved in each operation. When software engineers know a configuration scenario is composed of which products, and an operation requires which asset instances, they will be able to make selections based on their own preferences. The proposed variability at configuration scenarios, products, operations and asset instances levels necessitates the existence of a *constraints system* that controls the selection. For this purpose, we define a *derivation FM*, which is built according to the corresponding configuration.

Definition 7.2.1 (Derivation FM) :
We define a *derivation FM* as a constraint system represented in a feature model, offering a controlled selection of the operations to perform to achieve a derivation.

A generated *derivation FM* uses a classic FM formalism, but serves only in supporting the selection of the operations. The structural *features* of a *derivation FM* are the proposed *configuration scenarios* of a configuration, their corresponding *products*, the required *assets*, the possible *operations* to perform at each asset level and their corresponding *asset instances*. Configuration scenarios are considered as optional features, as long as more than one scenario is proposed. Similarly, their corresponding products are considered optional, except products that are part of all configuration scenarios are mandatory. The required assets are mandatory, since for each one of them an operation has to be selected. The operations corresponding to a required asset are represented as an *alternative group* of features, where only one operation for a required asset has to be selected. Respectively, if only one operation is proposed for a certain asset, it is represented as a mandatory feature. The asset instances of an asset are represented as an *or group* of features, since more than one instance can be required by a selected operation. If an instance is required by all possible operations of its asset, it is represented as mandatory. Similarly, if only one instance is proposed it is represented as mandatory. The *constraints* between the derivation FM features are constructed as follows:

- A *configuration scenario* implies an *operation* or an *or group* of *operations* (when a configuration scenario proposes more than one operation for a required asset).

- An *operation* implies an *asset instance* or an *and group* of *asset instances* (when an operation requires more than one asset instance for a required asset).

- An *asset instance* implies a *product* or an *or group* of *products* (when the asset instance is exploited by more than one product).

Definition 7.2.2 (Derivation scenario) :
We define a *derivation scenario* of a given configuration cf noted as $ds(cf)$, as the set of operations selected from the derivation FM to derive the desired product.

Selecting the operations to perform from derivation FM is a time-saving practice, since i.e. if a configuration scenario is selected, its corresponding operations and asset instances remain selectable, while the other ones are automatically disabled. Similarly, when an operation is selected, its corresponding asset instances are selected and its adjacent operations are deselected.

Example 20: Running example derivation FM

The generated derivation FM of the configuration cf_4 of the running example is shown in Figure 7.2. It represents the four possible configuration scenarios as optional features and there corresponding products. Since product p_2 is needed in the four configuration scenarios, it is a mandatory feature. For the assets $DeleteMatch.java$ and $match.jsp$, since there exists one possible operation that requires one asset instance, their corresponding operation and asset instance are mandatory. The assets $style.css$ and $SaveMatch.java$ have several possible operations, therefore, their possible operations are represented within an *alternative group*, enforcing the selection of only one operation. On the other side, their asset instances are represented within an *or group*, since an instance might be required by more than one operation. The asset instance $style.css^2$ is mandatory, since it is required by all the three possible operations of the asset $style.css$. The features "ConfigurationScenario", "Products", "Assets", the features corresponding to the assets names "DeleteMatch.java", "style.css", "match.jsp", "SaveMatch.java", and the features starting with "Operations_" and "Instances_" are structural level features.

7.2.2 Selection Factors

Our approach allows the selection of a configuration scenario and operations based on three factors: *numbers and indicators*, *developer preferences* and *cost estimation*.

7.2.2.1 Numbers and indicators

Software engineers can rely on numbers and indicators to select an appropriate configuration scenario. For instance, they might be interested in selecting the configuration scenario involving the least number of products or the configuration scenario that has the least number of operations that require an asset modification.

Example 21: Selection based on numbers and indicators

When achieving the configuration cf_4 of the running example, a software developer might be interested in selecting the configuration scenario cs_1 since it is composed of the least number of products (only p_2), or the configuration scenario cs_3 having the least number of operations that require a modification of assets.

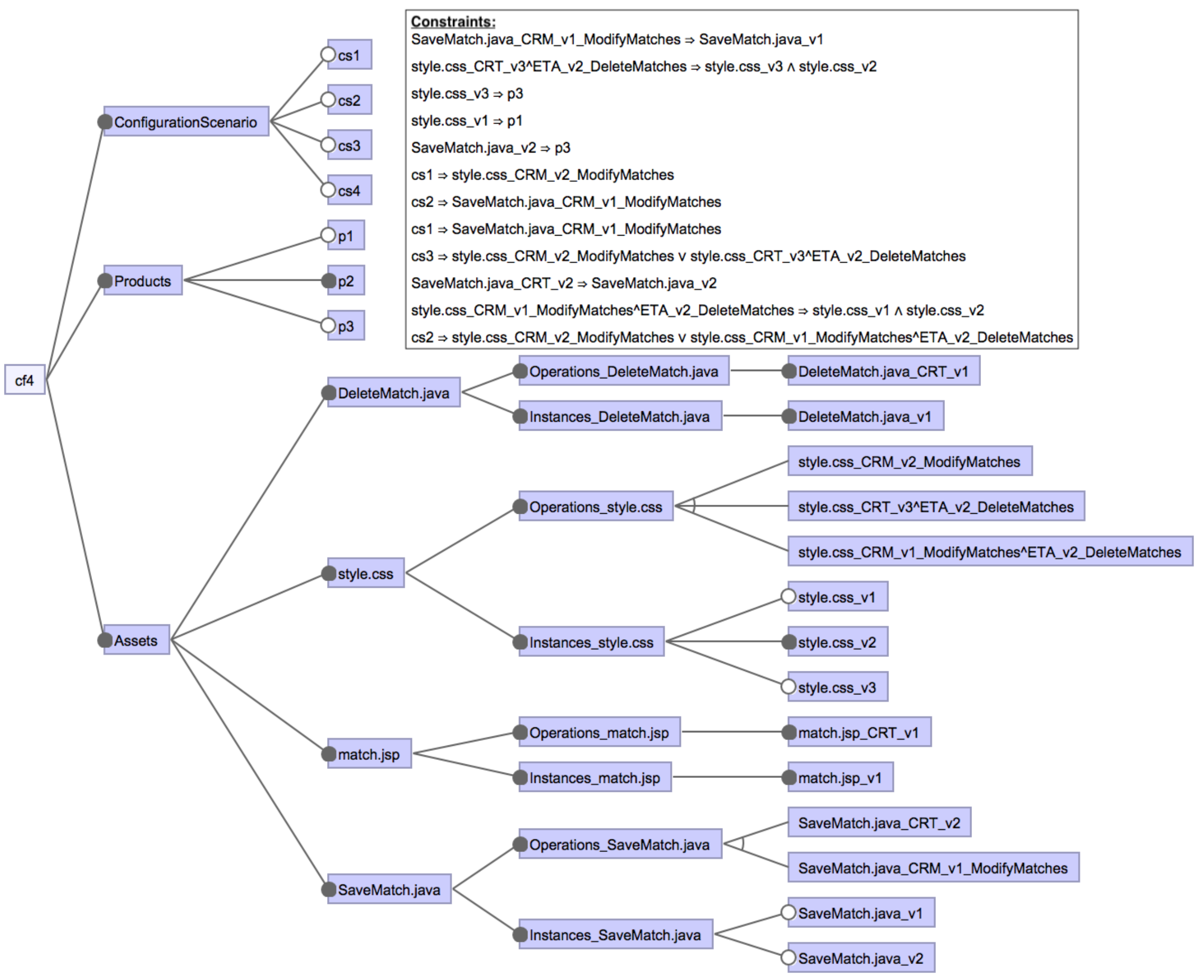


Figure 7.2: Running example cf_4 derivation FM

7.2.2.2 Developer Preferences

We also allow software developers to select the suitable scenario or operations to perform based on their personal preferences, according to their experience in developing the $\mathcal{SPL}$ product variants. A software developer might be interested in selecting the configuration scenario that is composed of the products that she is most familiar with. Moreover, the generated derivation FM allows to filter the operations based on the deselection of some undesirable products. Further, the derivation FM allows to filter operations by deselecting undesirable instances of assets whenever possible, such as old or untrusted instances. Thus, the constraints system allows not only to take the developer preferences into consideration, but also to reduce the number of decisions to be taken that simplifies the construction of the derivation scenario.

Example 22: Selection based on developer preferences

When achieving the configuration cf_4 of the running example, a software developer might be interested in working with configuration scenario cs_3 that involves p_3, which is a recently developed product, instead of working with cs_1 that involves p_1, which became an old product.

7.2.2.3 Cost Estimation

The second factor is to provide an estimated cost in terms of development effort and time for each operation and respectively for each configuration scenario. The cost estimation proposed in **Chapter 6** is an additional selection indicator that software developers can rely on. For instance, a software developer might select the configuration scenario having the least estimated cost. Similarly, she might select the operation having the least cost at each asset level, in order to compose the derivation scenario. Moreover, a software engineer has the possibility to select operations that do not necessarily belong to the same configuration scenario, as long as she selects one operation for each required asset. For instance, she can select the operations having the lowest estimated cost for all required assets, regardless if the operations belong to the same configuration scenario or not.

Example 23: Selection based on cost estimation

When achieving the configuration cf_4 of the running example, a software developer might select the configuration scenario $cs_3(cf_4)$ that has the least estimated cost (see Table 6.8) and is composed of less products compared to $cs_4(cf_4)$ having the same estimated cost. In this same concept, she might prefer the selection of the operation $\langle ETA, style.css^2, \{DeleteMatches\}\rangle$ for the asset $style.css$, since it has the least estimated cost (see Table 6.7) from the three possible operations of $style.css$.

7.3 Product Line Evolution

Despite that the main goal of our approach is to guide the derivation of new product variants from the $\mathcal{SPL}$, we consider that it is essential to permit the reuse of the newly derived products in future derivations. On the one side, a newly derived product variant can be requested in future, hence, it must be integrated in the $\mathcal{SPL}$, in order to be offered as one of the available ready-made product variants. On the other side, the features introduced by the newly derived products might be requested in future by other products, and therefore, their injection in the $\mathcal{SPL}$ feature models improves reuse. Similarly, the newly constructed asset instances resulting from the derivation are necessary to accomplish the implementation level of the injected features, as well as the existing ones.

The product line evolution is achieved through the following activities:

1. Product variant integration
2. Correlations and correlation indicators update
3. Feature models re-generation

It is important to mention that, product line evolution is necessary, only when a configuration is not achieved by integrality. In other words, a product resulting from a valid configuration is already provided by the $\mathcal{SPL}$, and hence, does not require a product line evolution. Otherwise, a configuration made in restrictive mode and introducing new features, or made in free mode, produces a new product that is not offered by the $\mathcal{SPL}$, and hence, requires a product line evolution to integrate it.

7.3.1 Product Variant Integration

Once a software developer composes the derivation scenario of the desired product, she is provided with the implementation files (asset instances) corresponding to the selected operations in the derivation scenario. At this level, the configuration FM is memorized, including if any, the new features added during configuration. Once the software developer fulfills the implementation of the desired product, the integration can be achieved. A product variant integration consists first of the addition of a new product p_j in the $\mathcal{SPL}$ products $\mathcal{P}$. Second, it consists in an automated identification of the newly added assets (if any) and asset instances.

Example 24: Integration of product p_4 of the running example

The product p_4 shown in Table 7.1 is the result of the configuration scenario cf_4. The integration of product p_4 in $\mathcal{SPL}$ led to the identification of a new asset instance $style.css^4$.

Table 7.1: Product p_4 asset instances

Product	Asset^{version}
p_4	match.jsp^1 SaveMatch.java^1 style.css^4 DeleteMatch.java^1

7.3.2 Feature Models Re-generation

Whenever a configuration leads to the derivation of a new product variant that is not provided by the $\mathcal{SPL}$, the feature models of the $\mathcal{SPL}$ (restrictive and free) require an update, regardless if the product variant introduces new features or implements a set of features that are not realized by an existing variant. As mentioned earlier, when a selected set of features during a configuration do not correspond to a valid configuration, the selected features are saved in a configuration FM, in order to be used to update the $\mathcal{SPL}$ feature models once the corresponding product is integrated.

To re-generate the restrictive FM, we perform a *FAMILIAR merge operation* [ACLF13] on the restrictive FM and the newly derived product FM, to obtain an updated restrictive FM, that integrates the configuration of the integrated product.

An update of the $\mathcal{SPL}$ restrictive FM implies a re-generation of the free FM, in order to remain consistent. However, the free FM is a constraint-free FM with all features optional except the root feature. Therefore, a re-generation of the free FM is performed only if the integrated product variant introduces new features that were not offered by the $\mathcal{SPL}$ prior to its derivation.

7.3.3 Correlations and Correlation Indicators Update

Integrating a new product variant into the $\mathcal{SPL}$ requires an update of the correlations between features and assets and between features and asset instances, in order to keep the correlations consistent with the $\mathcal{SPL}$ artifacts changes. Consequently, an update of correlations implies an

update of the correlation indicators that are used during cost estimation.

Example 25: FAMILIAR merge operation to re-generate $\mathcal{SPL}$ restrictive FM

Below are the FMs of the running example $\mathcal{SPL}$ and product p_4 in *FAMILIAR* language. A *FAMILIAR* merge operation re-generates the $\mathcal{SPL}$ restrictive FM.

Listing 7.1: Restrictive FM in FAMILIAR language before merge

```
fm_spl = FM (ManageMatches: AddMatches [ModifyMatches] [DeleteMatches];
DeleteMatches -> ModifyMatches;)
```

Listing 7.2: FM of product p_4

```
fm_p4 = FM (ManageMatches: AddMatches DeleteMatches;)
```

Listing 7.3: FAMILIAR merge operation over $\mathcal{SPL}$ restrictive FM and FM of p_4

```
fm_spl = merge sunion fm_spl fm_p4
```

Listing 7.4: Re-generated restrictive FM from FAMILIAR merge operation

```
fm_spl: (FEATURE_MODEL) ManageMatches: AddMatches [ModifyMatches] [DeleteMatches];
```

Example 26: Free FM eventual re-generation

Since the derivation of product p_4 of the running example did not introduce new features that were not provided by the $\mathcal{SPL}$, the free FM does not require a re-generation. However, if we consider a new product p_5 that introduces a new feature $Stats$, which is not offered by $\mathcal{SPL}$, the integration of p_5 would require a re-generation of the free FM.

7.3.3.1 Correlations Update

We provide an automated and incremental update of the $\mathcal{SPL}$ correlations. Correlations updates eventually touch the features implemented, assets employed, and asset instances exploited by the integrated product variant.

At feature to asset correlations level, three possible situations might occur:

— An addition of a new correlation.

— A removal of an existing correlation.

— A transformation of an *equivalence* correlation into an *implication* correlation.

At feature to asset instance correlations level, two possible situation might occur:

— An addition of a new correlation.

— A removal of an existing correlation.

The following rules must be applied in the given order.

An addition of a feature to asset correlation occurs in two conditions:

1. If the integrated product implements a new feature nf that was not implemented by any other product of $\mathcal{SPL}$ and employs a new asset na that was not employed by any other product, then a new feature to asset *equivalence* correlation is added: $nf \Leftrightarrow na$. Consequently, a new feature to asset instance correlation is added between nf and na^1, where na^1 is exploited by the integrated product and corresponds to the first asset instance of an.

2. If the integrated product implements a new feature nf that was not implemented by any other product of $\mathcal{SPL}$ and employs an existing asset ea that was already employed by another product, and the integrated product exploits a new asset instance ea^{n} of ea that was not exploited by any other product, then a new feature to asset *implication* correlation is added: $nf \Rightarrow ea$. Consequently, a new feature to asset instance correlation is added between nf and ea^{n}.

A removal of a feature to asset correlation occurs in the following condition:

1. If there exists a correlation between an existing feature ef and an existing asset ea, and the integrated product implements ef without employing ea, the correlation $c(ef, ea)$ is removed. Consequently, since none of the instances of ea is exploited by the integrated product, all feature to asset instance correlations between ef and the asset instances of ea are removed.

A transformation of an *equivalence* correlation to an *implication* correlation at a feature to asset correlation level occurs in only one condition:

1. If there exists an *equivalence* correlation between an existing feature ef and an existing asset ea, and the integrated product does not implement ef but still employs ea, and a new instance ea^{n} of ea is exploited by the integrated product, then, the *equivalence* correlation $ef \Leftrightarrow ea$ is transformed into an *implication* correlation $ef \Rightarrow ea$. Consequently, a new correlation is added between ea^{n} and each existing feature that ea is in correlation with and implemented by the integrated product.

Example 27: Running example correlations update

Referring to Table 7.1, the integration of the product p_4 in the $\mathcal{SPL}$ of the running example led to the addition of the following feature to asset instance correlations between the newly added asset instance $style.css^4$ and the features implemented by $p4$.

Table 7.2: Added correlations after integration of product p_4

Feature	**Asset$^{\text{instance}}$**
ManageMatches	style.css^4
AddMatches	style.css^4
DeleteMatches	style.css^4

Example 28: Running example correlation indicators update

The added correlations of Table 7.2 involve an update of the correlation indicators of the running example. We represent the added or updated entries of the correlation indicators tables in bold in the tables shown below. An entry for the asset instance $style.css^4$ is added to the asset instance correlated features table (Table 7.3) which is correlated to 3 features (the ones implemented by p_4).

Table 7.3: Updated running example asset instance correlated features after integration of product p_4

Asset$^{\textbf{Instance}}$	$\lvert F(a^{\mathbf{i}})\rvert$
$match.jsp^1$	2
$SaveMatch.java^1$	3
$SaveMatch.java^2$	2
$style.css^1$	3
$style.css^2$	4
$style.css^3$	2
style.css$^{\mathbf{4}}$	**3**
$DeleteMatch.java^1$	1

The feature correlated assets (Table 6.2) does not require any update since the integration of product p_4 did not require any update of correlations between features and assets.

The addition of a new instance of $style.css$ is reflected by the updated entry of this asset in the instances of assets (Table 7.4).

Table 7.4: Updated running example instances of assets after integration of product p_4

Asset	$\lvert AI(a)\rvert$
$match.jsp$	1
$SaveMatch.java$	2
style.css	**4**
$DeleteMatch.java$	1

The addition of the new correlations presented in Table 7.2 implies an update of their corresponding entries in the features correlated asset instances (Table 7.5).

Table 7.5: Updated running example features correlated asset instances after integration of product p_4

Feature	**Asset**	$\lvert AI(f/a)\rvert$
$ManageMatches$	$match.jsp$	1
$AddMatches$	$match.jsp$	1
$ManageMatches$	$SaveMatch.java$	2
$AddMatches$	$SaveMatch.java$	2
$ModifyMatches$	$SaveMatch.java$	1
ManageMatches	**style.css**	**4**
AddMatches	**style.css**	**4**
$ModifyMatches$	$style.css$	2
DeleteMatches	**style.css**	**2**
$DeleteMatches$	$DeleteMatch.java$	1

The updates of the correlation indicators illustrated above affect the correlation degrees, as shown in Table 7.6. New entries corresponding to the correlations between the added asset instance $style.css^4$ and the features in correlation with are added. In addition, some entries corresponding to the other instances of $style.css$ and the features in correlation with where updated.

Table 7.6: Updated correlation degrees between features and asset instances of the running example after integrating product p_4

Feature	**Asset$^{\text{instance}}$**	$cd(f, a^{\text{i}})$
ManageMatches	match.jsp^1	0.1666
AddMatches	match.jsp^1	0.1666
ManageMatches	SaveMatch.java^1	0.1111
AddMatches	SaveMatch.java^1	0.1111
ModifyMatches	SaveMatch.java^1	0.0833
ManageMatches	SaveMatch.java^2	0.1666
AddMatches	SaveMatch.java^2	0.1666
ManageMatches	style.css^1	0.1111
AddMatches	style.css^1	0.1111
ModifyMatches	**style.css^1**	**0.0833**
ManageMatches	style.css^2	0.0833
AddMatches	style.css^2	0.0833
ModifyMatches	**style.css^2**	**0.0625**
DeleteMatches	**style.css^2**	**0.0625**
ManageMatches	style.css^3	0.1666
AddMatches	style.css^3	0.1666
ManageMatches	**style.css^4**	**0.1111**
AddMatches	**style.css^4**	**0.1111**
DeleteMatches	**style.css^4**	**0.0833**
DeleteMatches	DeleteMatch.java^1	0.5000

7.3.3.2 Correlation Indicators Update

An update of a correlation implies an update in correlation indicators. The correlation degree between a feature f and an asset instance a^{i} noted $cd(f, a^{\mathrm{i}})$ is affected by four indicators that must be updated, in order to reflect the update on the correlation degree. These indicators are:

— The number of features correlated to the asset instance $|F(a^{\mathrm{i}})|$. This indicator is affected by the addition or removal of a correlation between the asset instance a^{i} and any feature.

— The number of assets correlated to the feature $|A(f)|$. This indicator is affected by the addition or removal of a correlation between the feature f and any asset.

— The number of instances of the asset $|AI(a)|$. This indicator is affected by an addition of a new instance of the asset a.

— The number of instances of the asset that the feature is in correlation with $|AI(f/a)|$. This indicator is affected by an addition or a removal of correlations between the feature f and the instances of the asset a.

7.3.4 Product Line Re-Definition

The migration process presented in **Chapter 4**, can occur during the lifetime of the product line. When refactoring shared assets is needed, or redefining existing feature models and their structure, software engineers can decide to re-create the product line by relaunching the migration process. Since this process is automated, it can done at any time without affecting the productivity. It is then a support for evolution of the $\mathcal{SPL}$.

7.4 Summary

With this chapter, we conclude the contribution part of our dissertation. Our contribution consisted on guiding the derivation of new product variants from an SPL based on C&O and evolving the SPL by integrating the newly derived variants into it. After defining the $\mathcal{SPL}$ and migrating its products and their artifacts in **Chapter 4**, we presented how we support the configuration of a desired product in **Chapter 5**, by identifying the possible configuration scenarios and operations to perform to achieve the derivation. In **Chapter 6**, we provided a cost estimation for the configuration scenarios and operations.

In this chapter, we presented how we support the derivation of the desired product, as a subsequent step after the configuration. Once the software engineer selects the required features for the desired product, we propose by means of a derivation FM the possible configurations and operations to perform at asset level. Hence, a software engineer selects the appropriate configuration scenario and operations based on indicators and number, her personal preferences and the cost estimation provided. We provide a reactive SPL evolution by integrating the newly developed PVs. A product variant integration consists of an automated identification of the new artifacts, an auto re-generation of the restrictive and free FMs and an automated and incremental update of the correlations and correlation indicators. Therefore, the new variants and their corresponding artifacts become subject to reuse in future derivations.

Part III

Implementation and Validation

CHAPTER 8

SUCCEED FRAMEWORK

Contents

8.1 Introduction **112**

8.2 Migration Process **112**

8.2.1 Product Line Initialization 112

8.2.2 Product Variants Supply 112

8.2.3 Restrictive FM supply 112

8.2.4 Free FM generation 113

8.2.5 Assets and asset instances identification 113

8.2.6 Correlations Identification 114

8.3 Configuration Process **115**

8.4 Derivation Process **117**

8.5 Evolution Process **119**

8.6 Summary **120**

8.1 Introduction

In this chapter, we describe the implementation of our approach. We implemented our approach by developing a framework called **SUCCEED** that stands for *SUpporting Clone-and-own with Cost-EstimatEd Derivation*. SUCCEED allows to migrate a set of product variants into an SPL to support the configuration and derivation of a new variant and to permit an enrichment of the SPL with the newly derived variants. The time this dissertation was written, SUCCEED was partially implemented with a graphical user interface, while other modules of its implementation where available only with a command line interface. Hence, we aim to provide in future a complete graphical user interface support for SUCCEED that covers all its modules. SUCCEED is developed using Java technology, based on a Java web service, and the graphical interface consists of a web application developed mainly using JavaScript AngularJS framework.

8.2 Migration Process

8.2.1 Product Line Initialization

In order to integrate existing PVs, the name of the $\mathcal{SPL}$ to be created and the path under which it will be located are specified by a software engineer. Under this path, SUCCEED specifies a *git* repository [CS14], in which the migrated product variants will be pushed.

Ignoring files from being tracked is a useful functionality provided by git. SUCCEED allows to specify the files and directories to be ignored, such as specific type of files and external libraries, which do not belong to the product line artifacts.

8.2.2 Product Variants Supply

The existing product variants to be migrated are given as an input to SUCCEED. For each PV, a unique name has to be assigned to it, and the repository in which its artifacts are located has to be specified. In addition, an FM representing the features that it implements has to be provided. A *git branch* [CS14] is created for each migrated PV, and the product name is assigned to the branch name. The implementation files corresponding to a PV are committed within its branch. As a result, the product line repository will consist of n branches and n commits (one commit per branch) where n is the number of existing product variants. The adopted git structural definition of the $\mathcal{SPL}$ enables a simple retrieval of an existing product, whenever it is required by a configuration. Moreover, it provides an independent evolution of each variant within its branch, when this evolution consists of a maintenance of the implementation code and not an update of the variant at specification level (features).

8.2.3 Restrictive FM supply

After supplying the PVs with their corresponding FMs, a global FM of the product line is generated by SUCCEED, as the *FAMILIAR* merge [ACLF13] of the PVs FMs. This is what we call in our approach the *restrictive FM*. The restrictive FM provides an abstract representation

of the supplied PVs in terms of business features. It has to allow the configuration of the exact set of the migrated PVs. A structural defect in the provided FMs of the PVs or of the restrictive FM prevents the initialization of the $\mathcal{SPL}$. For instance, this might result of PVs not having a common root feature in their corresponding FMs.

8.2.4 Free FM generation

To enable configurations that break the constraints imposed by the restrictive FM, the *free FM* has to be generated. We defined an algorithm that applies the following operations on a copy of the *restrictive FM*:

1. Remove all *constraints*

2. Remove a *group (or, alternative)* and connect its children features directly to the parent of the group as optional features

3. Transform a *mandatory* feature into an *optional* feature (except for the root feature)

Since the free FM has all its features optional (except the root) and has constraints, any set of selected features including the root can be selected during configuration, allowing a complete reuse at features level.

8.2.5 Assets and asset instances identification

From the product variant base, SUCCEED identifies the common artifacts of several products, those that are unique and those that exhibit variations. We characterize as "assets", the file path of the artifacts and as "asset instances" the artifacts themselves (implementation files). Thus to the same asset can correspond to several asset instances (implementations).

Algorithm 1 describes the mechanism that we adopt to define the $\mathcal{SPL}$ products and extract assets and their corresponding asset instances from the PVs. The algorithm loops over the implementation files of each PV. For each PV we define a corresponding product in $\mathcal{SPL}$. For each file in a PV, if it does not exist an asset in $\mathcal{SPL}$ assets ($\mathcal{A}$) that has the same path name as the file, a new asset is created and the path name of the file is assigned to it. Then, a new asset instance is created, an instance number 1 is assigned to it, in addition to the path corresponding to its implementation file. The asset instance is registered as an instance of the asset (being the first in this case), and registered as one of the instances exploited by the product created during the current iteration. As well, the asset is registered as one of the $\mathcal{SPL}$ assets. Otherwise, if there exists an asset having the same path name, an additional test is done to check if one of its instances has the same implementation of the file. We consider that two implementations are equal if their content is similar, regardless if their string values are equal [CS14]. If so, the identified instance is appended to the list of asset instances exploited by the product. Otherwise, a new instance is created, assigned a new instance number (computed from the number of instances that the asset already has), in addition to the path corresponding to its implementation file. As well, the asset instance is appended to the instances of the identified asset, and to the asset instances exploited by the product.

Algorithm 1 Assets extraction and products definition

Input: $PV\{files\}$ a set of Product Variants
Output: $\mathcal{A}$ a set of Assets, $\mathcal{P}$ a set of Products

```
1: Initialize: A = {}, P = {}
2: for all pv ∈ PV do
3:   new p
4:   for all file ∈ files(pv) do
5:     if ∄ (a ∈ A ⋀ name(a) = name(file)) then
6:       new a
7:       name(a) ← name(file)
8:       new a^i
9:       instanceNo(a^i) ← 1
10:      implementation(a^i) ← implementation(file)
11:      AI(a).append(a^i)
12:      AI(p).append(a^i)
13:      A.append(a)
14:    else
15:      if ∃ (a^i ∈ AI(a) ⋀ implementation(a^i) = implementation(file))* then
16:        AI(p).append(a^i)
17:      else
18:        new a^i
19:        instanceNo(a^i) ← size(AI(a)) + 1
20:        implementation(a^i) ← implementation(file)
21:        AI(a).append(a^i)
22:        AI(p).append(a^i)
23:      end if
24:    end if
25:  end for
26:  P.append(p)
27: end for
```

*Two implementations are equal if the *diff* between them is null

8.2.6 Correlations Identification

Algorithm 2 represents how correlations are identified. After identifying the $\mathcal{SPL}$ products $\mathcal{P}$, features $\mathcal{F}$ collected from the products FMs, assets $\mathcal{A}$ and their corresponding instances, SUCCEED identifies the $\mathcal{SPL}$ correlations $\mathcal{C}$. As presented in Algorithm 2, for all assets $\mathcal{A}$, for each $a \in \mathcal{A}$, we loop on all features of $\mathcal{F}$. A correlation between an asset a and a feature f holds each product implementing f employs a and each product employing a implements f, hence if $P(a) = P(f)$, or if each product implementing f employs a, hence $P(a) \subset P(f)$ and for each instance a^{i} of a exploited by a product implementing f, a^{i} is not exploited by any product that is not implementing f. When a correlation holds, if $P(a) = P(f)$ the correlation type is an equivalence, otherwise it is an implication. Each feature to asset correlation $c(f, a)$ is appended to the $\mathcal{SPL}$ correlations $\mathcal{C}$. For each identified feature to asset correlation, the instances of the asset must be evaluated to determine if they are in correlation with the feature. A feature to asset instance correlation holds if there already exists a feature to asset correlation $c(f, a)$, and there exists at least a product that implements f and exploits a^{i}. Moreover, each feature to asset instance correlation is appended to the $\mathcal{SPL}$ correlations $\mathcal{C}$.

If an artifact (feature, asset, or asset instance) has no correlations, the $\mathcal{SPL}$ is not considered *complete*. For this reason, SUCCEED displays a warning informing that the tool does not guarantee the support and evolution of the $\mathcal{SPL}$.

Algorithm 2 Correlations identification

Input: $\mathcal{P}$ Products of $\mathcal{SPL}$, $\mathcal{A}$ Assets of $\mathcal{SPL}$, $\mathcal{F}$ Features of $\mathcal{SPL}$
Output: $\mathcal{C}$ Correlations of $\mathcal{SPL}$

```
1:  Initialize: C = {}
2:  for all a ∈ A do
3:      for all f ∈ F do
4:          if [P(a) = P(f)] ⋁ [P(a) ⊂ P(f) ∧ ∀a^i ∈ AI(P(f)), a^i ∉ AI(P(a) \ P(f))] then
5:              new c(f, a)
6:              if P(a) = P(f) then
7:                  c(f, a).type ← equivalence
8:              else
9:                  c(f, a).type ← implication
10:             end if
11:             C.append(c(f, a))
12:             for all a^i ∈ AI(a) do
13:                 if ∃ p, f ∈ F(p), a^i ∈ AI(p) then
14:                     new c(f, a^i)
15:                     C.append(c(f, a^i))
16:                 end if
17:             end for
18:         end if
19:     end for
20: end for
```

8.3 Configuration Process

SUCCEED provides two configuration modes. Figure 8.1 shows a configuration in *restrictive mode* and Figure 8.2 shows a configuration in *free mode*. When a new configuration is initiated, the default configuration mode is *restrictive mode*. In a restrictive mode, mandatory features are selected automatically and cannot be deselected, as shown in Figure 8.1 i.e. features $Rounds$, $Stages$, $Groups$, $Standing$. In contrary, optional features, such as $Players$, $Replays$, $Calendar$ and $News$, are by default unselected, and can be selected (i.e. $Calendar$ and $News$) or deselected (i.e. $Replays$). A deselection of a parent feature, implies a deselection of its children features. Further, depending on the cross-tree constraints of the restrictive FM, a selection of a feature might imply an automatic selection or deselection of other features in the FM configuration i.e. if no product implements both $Players$ and $Replays$, selecting the feature $Players$ will automatically deselects the feature $Replays$.

Switching from restrictive mode to free mode during a configuration can be made with a single click. A first-time switch from restrictive to free mode takes the actual state of the configuration in restrictive mode as an initial state during the free mode. In other words, all selected

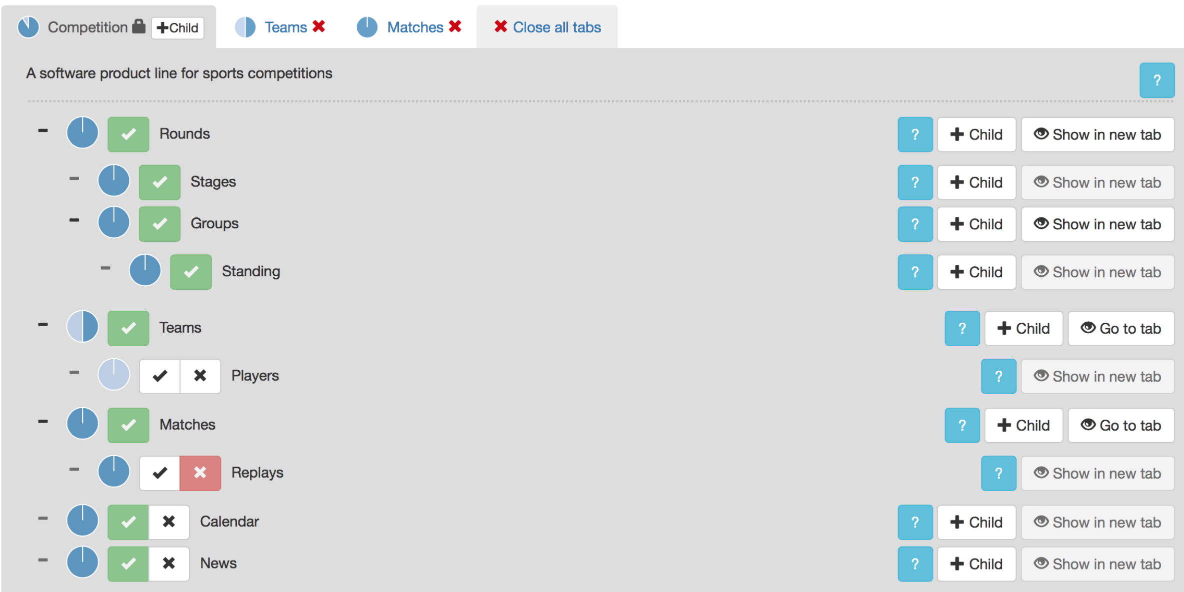


Figure 8.1: An example of a configuration in restrictive mode

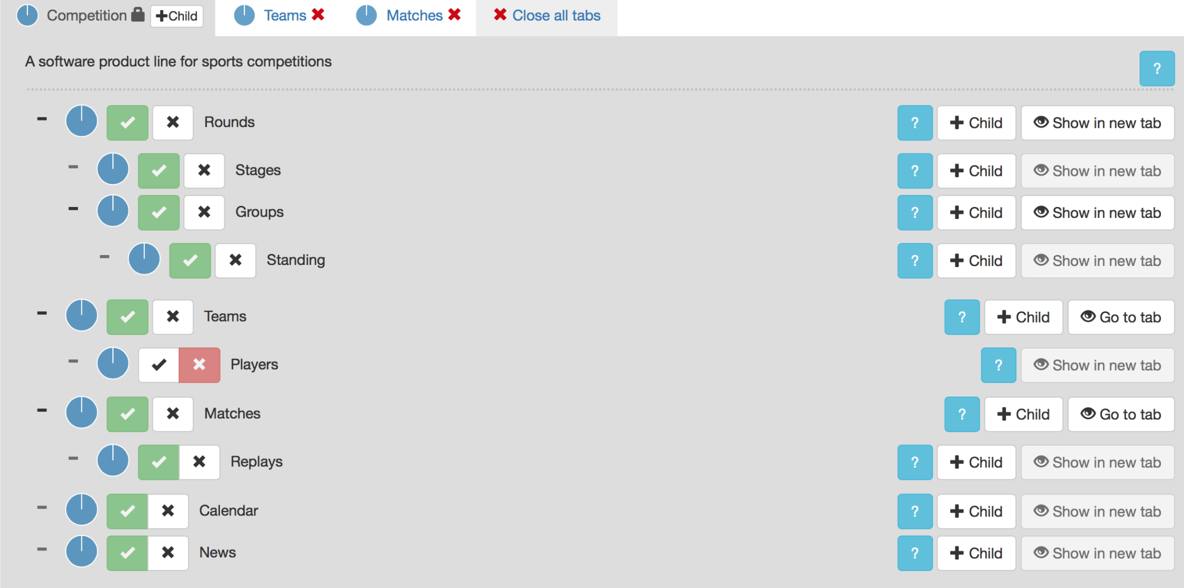


Figure 8.2: An example of a configuration in free mode

features are selected in free mode, and all deselected features are deselected in free mode. Since no mandatory features in free mode except the root feature, all features can be deselected. As shown in Figure 8.2, the feature *Rounds* i.e. which is initially a mandatory feature in restrictive mode can be deselected in free mode since it became an optional feature. Similarly to restrictive mode, if a parent feature is deselected its children features are automatically deselected, hence, deselecting *Rounds* will automatically deselect *Stages*, *Groups* and *Standing*. In contrary to restrictive mode, no constraints are present in free mode, hence, selecting or deselecting a feature affects only its children features. Since all features except root are optional in free mode, we provide two button actions to facilitate the selection/deselection of features. The first option allows to auto-select all unselected features, and the second allows to auto-deselect all unselected features. Hence, a software engineer can either select all required features and use the

auto-deselect action to deselect all other unrequired features, or deselects all unrequired features and use the auto-select action to select all required features. Recall that those two actions are available only in free mode.

Switching back to restrictive mode is again possible. This action saves the actual state of the free mode and switches back to the last state of the restrictive mode before the switch made to free mode. Thus, a current configuration can have two independent flows in each mode.

One of the important functionalities that we introduced in our approach is the possibility of adding new features during the configuration. This functionality is available in both restrictive and free mode. A new feature can be added only as a child for a selected feature i.e. as show in Figure 8.1 a child feature cannot be added for the feature $Players$ which is currently unselected neither for the feature $Replays$ which is currently deselected. A feature added in restrictive mode is not added automatically in free mode, except for the first-time switch from restrictive to free mode. Similarly no two features can have the same name. It is possible to add a feature in restrictive mode and adding it with the same name later on in free mode, however, it is prohibited to add two new features with same name in same mode, or also adding a new feature with the name of an existing one. A feature added during a configuration is not visible in other configurations. It becomes visible only if the configuration was achieved by a derivation of a new product and the product is integrated into the $\mathcal{SPL}$.

The configuration process is accomplished by selecting all the required features, deselecting all the unrequired ones and adding the new features if any. Once done, the configuration is saved and the selected features are memorized. In addition, the new features added to the configuration are memorized. Next is the derivation process where operations to perform to derive the desired product are to be selected.

8.4 Derivation Process

The derivation process is dependent on the configuration process. If the configuration is done in restrictive mode, this means that an existing product implements all and only the required existing features for the desired product. Hence, SUCCEED proposes the implementation files of this product as a solution and mentions the need of integrating the new features added during configuration, if any, in the proposed files. Hence, if new features have to be added, our approach does not provide any guidance concerning which assets are suspect to modification to integrate the new features.

If the configuration is done in free mode, SUCCEED proposes the possible configuration scenarios and operations, in addition to their corresponding estimated costs, based on the generated *derivation FM*. Figure 8.3 shows the SUCCEED interface corresponding to the derivation FM of the running example. Recall that the derivation FM is automatically generated according to the automatically identified configuration scenarios and operations as a subsequent step to the configuration process.

According to Figure 8.3, the interface displays the four possible configuration scenarios of the running example with their corresponding estimated cost, where only one configuration can be selected. Recall that the default estimated cost corresponding to each configuration scenario,

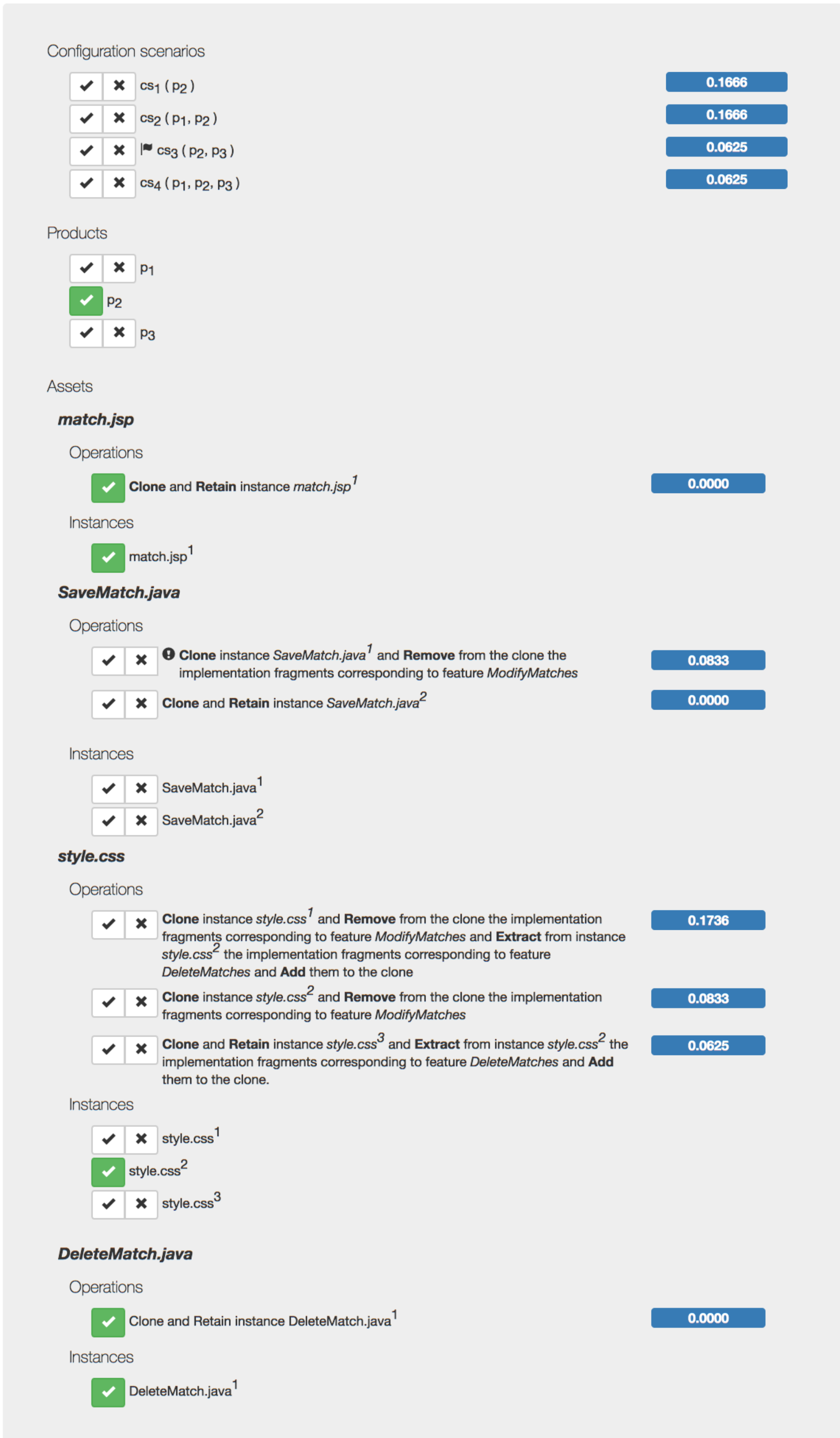


Figure 8.3: Running example derivation process in SUCCEED

prior to operations selection, is the sum of its lowest cost operation for each required asset. This value might be updated upon the selection of operations, i.e. in case an operation having a higher cost was selected for a required asset. A flag icon (see configuration scenario cs_3) is displayed next to the recommended configuration scenario, the one having the lowest estimated cost. The configuration scenarios cs_3 and cs_4 have the same estimated cost, since the operations having the lowest cost at each asset level are similar for both. However, cs_3 is given the recommended configuration scenario title, because its products are subset of the ones of cs_4.

Product p_2 is displayed as mandatory, because all the proposed operations require an asset instance exploited by p_2. Deselecting an optional product implies an automatic deselection of the asset instances exploited only by it, in addition to the operations that use those instances. For example, deselecting p_1 implies a deselection of instance $style.css^1$ which is not exploited by another product, and a deselection of the first operation of $style.css$ which uses this instance.

For an asset having one operation, the operation is displayed as mandatory, similarly to the instances that it uses. The assets $match.jsp$ and $DeleteMatch.java$ are two examples of this situation.

Once a configuration scenario is selected, all operations that do not correspond to it are automatically deselected. For example, if cs_3 is selected, the first operation of $style.css$ is automatically deselected since it uses the instance $style.css^1$ that is not implemented by any of the products p_2 and p_3 of cs_3. Furthermore, the operations composed of a single CRT action corresponding to the selected configuration scenario are automatically selected since their estimated cost is 0. For instance, when cs_3 is selected, the CRT operation of $SaveMatch.java$ is automatically selected, and consequently, its selection implies an automatic deselection of the first operation of $SaveMatch.java$, since only one operation can be selected per asset.

Moreover, if more than one operations are proposed for a required asset, and one the proposed operations is a single CRT operation, an exclamation mark icon is displayed near the other proposed operations for the asset, to represent a warning that there already exists an asset instance implementing all and only the required existing features. The first operation proposed for $SaveMatch.java$ represents this situation, where this operation proposes the construction of a new instance, that is supposed to be similar to the already existing instance $SaveMatch.java^2$.

Furthermore, deselecting an asset instance implies an automatic deselection of the operations that use the deselected asset instance. For example, deselecting $style.css^1$ for considering it an old undesired instance for derivation, implies an automatic deselection of the first operation of $style.css$ that uses this instance.

Finally, for each required asset, one operation has to be selected. Recall that the operations selection can be made regardless the configuration scenarios they belong to, by selecting operations that belong to different configuration scenarios.

8.5 Evolution Process

Once the derived product is constructed, it has to be integrated in SUCCEED in order to permit its systematic reuse and respectively the reuse of its artifacts for deriving new variants.

To integrate the derived product, its implementation files are provided. Hence, a new branch is created in the git repository that contains the $\mathcal{SPL}$ products, where the name of the product is used as the branch name. The provided implementation files are committed within the created branch.

The integration of a new product implies an automated discovery of new assets and asset instances, and an automated update of the $\mathcal{SPL}$ correlations and correlation indicators, as explained in **Chapter 7**. The identification of a new file (a file that was not employed by any other existing product) in the provided implementation files of the derived product, leads to the identification of a new asset and a new instance corresponding to it. The identification of new instances of an asset is done by performing a *diff* operation to compare a certain file with the existing instances having the same name.

To permit the reuse of the integrated product and its corresponding artifacts in future derivations, SUCCEED performs an automated update of the restrictive and free FMs as explained in **Chapter 7**. Consequently, the features added during the configuration of the integrated product if any, become visible in new configurations. Further, the selection of the set of features implemented by the integrated product refers now to a valid configuration that implies an automated derivation of the newly integrated product.

8.6 Summary

In this chapter, we presented an overview of the SUCCEED framework that implements our approach. The SUCCEED framework covers the life cycle of our approach. It first allows the migration of existing product variants into an SPL. Second, it permits the generation of new configurations in both restrictive and free modes. Third, it supports the derivation with the possible configuration scenarios and operations, as well as their estimated cost. Finally, it enables the enrichment of the SPL with the newly derived products.

Nowadays, not all modules of SUCCEED are supported with a graphical user interface. Therefore, we aim to provide graphical interface to the command line interface modules of the approach and connect its modules in a standalone framework.

CHAPTER 9
APPROACH VALIDATION

Contents

9.1 Introduction 122
9.2 Validation 122
9.2.1 Experiments 122
9.2.2 Results analysis 123
9.2.3 Overtaking Challenges 124
9.3 Limitations 125
9.4 Threats to Validity 125
9.5 Summary 126

9.1 Introduction

We validate our approach in this chapter based on experiments made on a case study consisting of a family of 8 product variants. We measure and analyze indicators corresponding to the configuration, derivation and integration of new product variants, in order to evaluate the effectiveness of our approach. In addition, we present in this chapter some limitations of our approach and threats to validity.

9.2 Validation

9.2.1 Experiments

We achieved the validation by analyzing statistical information that we collected upon the configuration and the incremental derivation of 5 new variants and their incremental integration in the case study SPL that consists initially of 3 variants. By incremental derivation and integration, we mean that each newly derived variant is integrated in the SPL, the SPL restrictive and free FMs are auto re-generated and the correlations are updated prior to the derivation of another variant. Hence, each newly derived variant becomes a support element during the upcoming derivation. The SPL comprises when it has its 8 PVs a total of 93 features, 271 assets and 296 asset instances with an average of 66 features, 214 assets and 4.7KLOCs per PV.

Table 9.1 shows the number of features added during each configuration and the number of assets and asset instances added to the SPL after the derivation of the PV corresponding to each configuration. In order to evaluate the effectiveness of our approach, we measured and analyzed significant indicators (see Figure 9.3) concerning the number of configuration scenarios and operations of the 5 configurations of the case study.

Table 9.1: Metrics of 5 sequential configurations to derive new PVs

Configuration	cf_4	cf_5	cf_6	cf_7	cf_8
Number of features added by the configuration	0	0	25	0	0
Number of Assets added after derivation	0	0	8	0	24
Number of Asset instances added after derivation	3	1	11	1	25

The 8 PVs of the case study correspond to a family of Web applications developed to post news, media, and results of a soccer competition[1]. Each PV consists of a restful web service written in Jersey, a database structure built in MySQL, and client-side Web interfaces written in HTML, CSS and JavaScript frameworks including AngularJS, JQuery, UI-Router, Bootstrap and others. Table 9.2 shows the main features implemented by the 8 product variants.

Figure 9.1 and Figure 9.2 show some graphical interfaces of products p_7 and p_8. Product p_7 corresponds to the Champions League 2016 competition, while product p_8 corresponds to EURO 2016 competition. Since feature $Matches$ is implemented by both p_7 and p_8, their corresponding interfaces look similar (see Figure 9.1a and Figure 9.2a). The interface displaying match details defers between p_7 and p_8, since the former implements feature $Match\,facts$ while

[1]The case study products do not correspond to the running examples products, which were demonstrated throughout the dissertation.

Table 9.2: Main features of case study product variants

Features	Products							
	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
Competition	✓	✓	✓	✓	✓	✓	✓	✓
Matches	✓	✓	✓	✓	✓	✓	✓	✓
Rounds	✓	✓	✓	✓	✓	✓	✓	✓
Teams	✓	✓	✓	✓	✓	✓	✓	✓
Players	✓	✓			✓	✓	✓	✓
News		✓	✓		✓			✓
Match facts		✓				✓	✓	✓
Match stats						✓		✓

the latter implements $Match\,facts$ and also $Match\,stats$, which corresponds to the statistical tables displayed below the match facts (see Figure 9.1b and Figure 9.2b). Further, since product p_7 does not implement the feature $News$, the "News" item is not part of its menu. On the contrary, product p_8 implements the feature $News$ (see Figure 9.2c) and the item "News" is part of its menu.

9.2.2 Results analysis

As shown in Figure 9.3a, the number of configuration scenarios per configuration increases considerably whenever the family of software products becomes richer. This is due to the injection of the newly derived products in the SPL, that become candidates for derivation of other products. Respectively, Figure 9.3b shows that the average number of products involved in a configuration scenario increases too. Identifying the possible configuration scenarios for a given configuration is time-consuming if done manually, especially when the SPL becomes mature, since it is proportional to the number of products that partially implement the required features. Our approach resolves this difficulty by automating the identification of the possible configuration scenarios. When the number of products involved in a configuration scenario becomes elevated, software engineers aim to prioritize the possible configuration scenarios based on preferences over products, i.e. by preferring configuration scenarios that involve the products that they are most familiar with. The generated derivation FM provided by our approach satisfies this need by allowing software engineers to filter the possible configuration scenarios by deselecting undesired products.

Figure 9.3c shows that the number of required existing assets per configuration is high for the case study with an average of 88%. In other words, each newly derived product is constructed by reusing around 88% of the SPL assets. This indicator reflects the degree of reuse that SPLs can offer in general and our approach in particular. Identifying the required assets for a product derivation can be a tedious and time-consuming task when done manually, especially when the SPL consolidates a large amount of assets. Our approach copes this problem by providing an automated discovery of the required assets for derivation.

Despite that the number of required assets can be large, the number of assets to modify might be few, same as shown in Figure 9.3d, where for the case study the average number of assets to modify is only 2%. This indicator might widely vary in other case studies, depending on the

variability level it provides, as well as the features interaction level, reflected in the way in which the assets are constructed. Regardless if the number of assets to modify is low or high, software engineers have to identify between the required assets, which ones are to be modified to achieve the derivation. This step is expensive if done manually, since it requires a content check of the asset instances of each required asset. In contrary, our approach automatically identifies the assets that require modification, based on the automatically defined operations at each asset level. Hence, software engineers recognize which asset instances have to be cloned and retained for derivation without being modified and which asset instances require to be cloned and modified. Figure 9.3e shows that the average number of operations per asset for each configuration is very low in this case study. This is due to the low number of assets to modify, where for most assets the only proposed operation consists of a single clone and retain (CRT) action. For the assets that have several proposed operations, the provided cost estimation at operation level facilitates the choice of a favorable operation. Moreover, the selection of a favorable operation can be made based on the asset instances involved in the operation, since the derivation FM specifies the asset instances of each operation.

Finally, the elevated number of possible configuration scenarios makes difficult the choice of a favorable scenario for derivation. However, the cost estimation provided by our approach can facilitate the choice of a favorable scenario. Figure 9.3f shows that the coefficient of variation between the estimated cost of the configuration scenarios ranges between 36% and 99% with an average of 64%. This means that the variation between the estimated cost of the configuration scenarios is elevated, which can facilitate the choice of a favorable scenario. Moreover, the elevated coefficient of variation reflects that if software engineers select the configuration scenarios having the lowest estimated cost, they are supposed to save a considerable amount of time and effort during derivation.

9.2.3 Overtaking Challenges

Challenge 1.a consisted on identifying mappings between features and assets. The average number of assets to modify per configuration scenario (see Figure 9.3d) decreases as long as new products are integrated in the SPL. This is due to the purification of the identified correlations. Since correlations are updated each time a new variant is integrated in the SPL, they get a better level of precision, and therefore, the estimated number of products to modify decreases and the proposed operations become more precise.

Challenge 1.b consisted on identifying the possible configuration scenarios and asset level operations. Our experiments on the case study verified that all possible configuration scenarios and operations where identified and since this step is automated by our approach, it saves a large amount of time compared to a manual identification process.

Challenge 1.c consisted on facilitating the choice of configuration scenarios and asset level operations. Figure 9.3a shows that the number of possible configuration scenarios increases accordingly to the maturity of the SPL. This reflects the necessity of facilitating their selection. As shown in Figure 9.3f, the coefficient of variation between the estimated cost of the possible configuration scenario is high, which facilitates the selection. Moreover, the user preferences options that we proposed contribute in facilitating the selection.

Challenge 2.a consisted on helping the definition of new features when adding new variants and structuring the SPL features after integration. Table 9.1 shows that 25 features where added to derive product p_6. Moreover, those features where reused in the derivation of the new products (p_7 and p_8).

Finally, **Challenge 2.b** consisted on integrating the newly derived products into the SPL. As shown in Figure 9.3a and Figure 9.3b, the new products where used as part of the proposed configuration scenarios for future derivations. As well, the new features that they implemented where reused in the derivation of new products.

9.3 Limitations

A limitation of our approach is that it is dependent on the architecture of the developed SPL. A change in structure or naming of the SPL artifacts affects the identified correlations. However, adhering to the proposed operations during product derivation avoids such inconsistencies. Further, our approach permits to reconstruct the $\mathcal{SPL}$ in case a refactoring is needed, without extra efforts or loss of information.

Another limitation is that asset instances are identified at file level, while several related works when performing feature identification or feature location [FLLHE14, Mar16], map features to implementation blocks of several files. Such techniques can be complementary to our approach, since we consider that guidance is the most meaningful when provided at file level.

Moreover, our approach does not propose any guidelines regarding the modification of code fragments inside files. Therefore, the adapters employed in [FLLHE14] and [Mar16] to identify the blocks of artifacts can be used to provide this guidance, especially when the files of the migrated products correspond to artifact types to which adapters are provided. Similarly, the hints proposed by [FLLHE14] when product completion is required to achieve the derivation improve the guidance that can be proposed.

Finally, in our approach we did not take into consideration organizational factors such as coordination between development team members, development phases, criteria selection and prioritization, product release, and customer feedback [BBS11, BWW+18]. Similarly, we did not integrate international distortions in our approach when defining our cost estimation functions [MBF12]. We consider such factors beyond the core of our approach, especially when our approach aims only to provide guidance without imposing derivation solutions.

9.4 Threats to Validity

Conclusion Validity In our experiments, and based on the polyglot nature of the software products we are working on, we concluded that increasing number of products in the SPL increases the number of possible configuration scenarios (see Figure 9.3a). However, in some architectures where features are not propagated on several products, this conclusion might become wrong. Whereas, in our experiments we did not identify a relationship between the average number of products in a configuration scenario (see Figure 9.3b) and the average number of possible operations per asset (see Figure 9.3e). Despite that the graph indicators show this relationship for the case study, we consider that it might not be the case for other case studies.

Internal Validity In our case study, we considered that one of the reasons of the elevated number of required assets (see Figure 9.3c) is due to the architecture of polyglot systems, where feature interaction is elevated. However, other unidentified factors might also affect the number of required assets, especially for case studies of different architectures employing different artifact types.

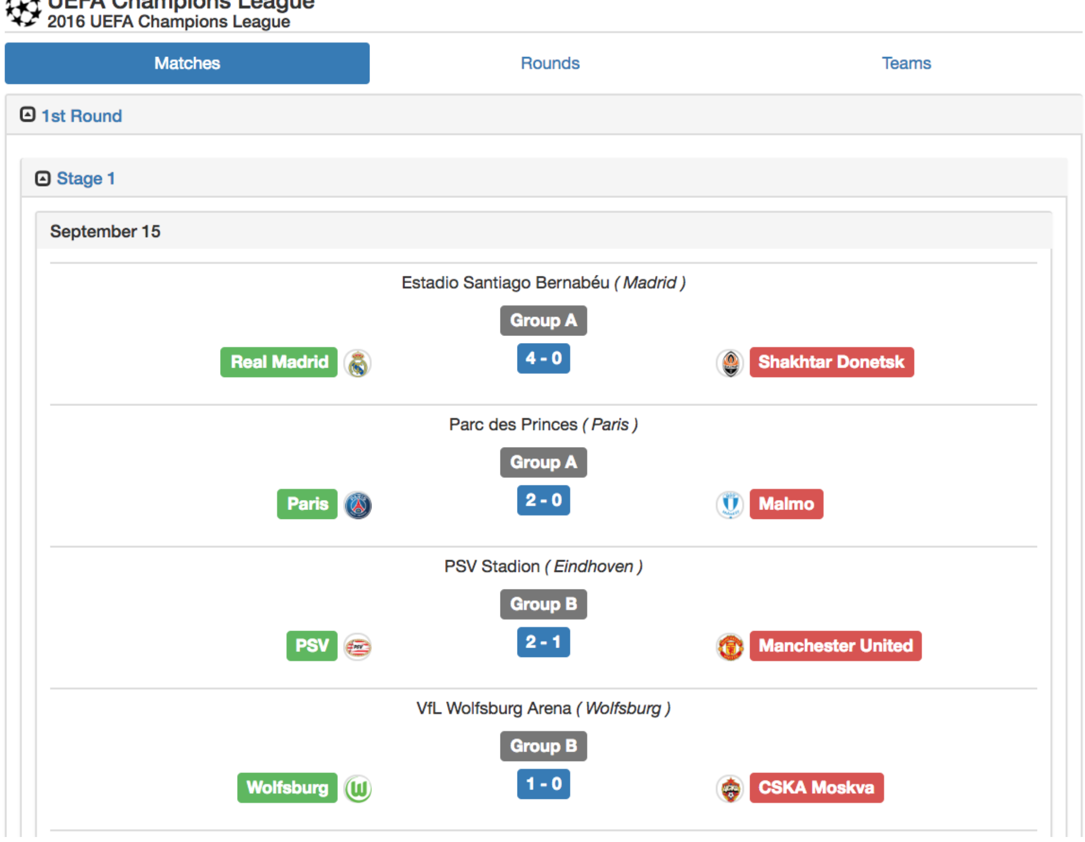
Construct Validity The experiments were made on a case study that we developed. Indeed, due to our limited time and resources, we could not achieve experiments on industrial case studies, to measure the effectiveness of our approach on real industrial situations. If done, it can bring valuable indicators that show the variation between the time spent by software engineers to manually accomplish tasks such as identifying configuration scenarios, required assets, assets to modify and operations, compared to the automation that our approach provides of those tasks.

External Validity It is clear that the order in which the configurations and respectively the derivation and integration of new products are made produces different outputs of the experiments results. Our approach guarantees for the experiments that we made, that no matter what is the order in which the products are derived and integrated in the SPL, the final state of the SPL in terms of features models, identified artifacts and correlations is the same. Since we did not perform experiments on all kinds of software families, we are not able to validate that our assumption can always be true. Similarly, the outcome of our experiments corresponds to the structure of the case study, hence, another case study having different architecture such as monoglot systems, and using other artifact type such as images, might produce different results.

9.5 Summary

In this chapter, we validated our approach on a real case study to demonstrate its effectiveness. The indicators of Figure 9.3 reveal the importance of providing the necessary support to software engineers to accomplish the derivation of new products. Our approach automates several time-consuming and tedious tasks, such as the identification of the possible configuration scenarios and required assets, the identification of the assets that require modification, and the operations to perform to achieve the derivation. Moreover, our experiments show that the generated derivation FM for each configuration facilitates the choice of the favorable configuration scenario and operations, by providing an environment allowing software engineers to make their choices based on their own preferences and based on the cost estimation of the configuration scenarios and operations.

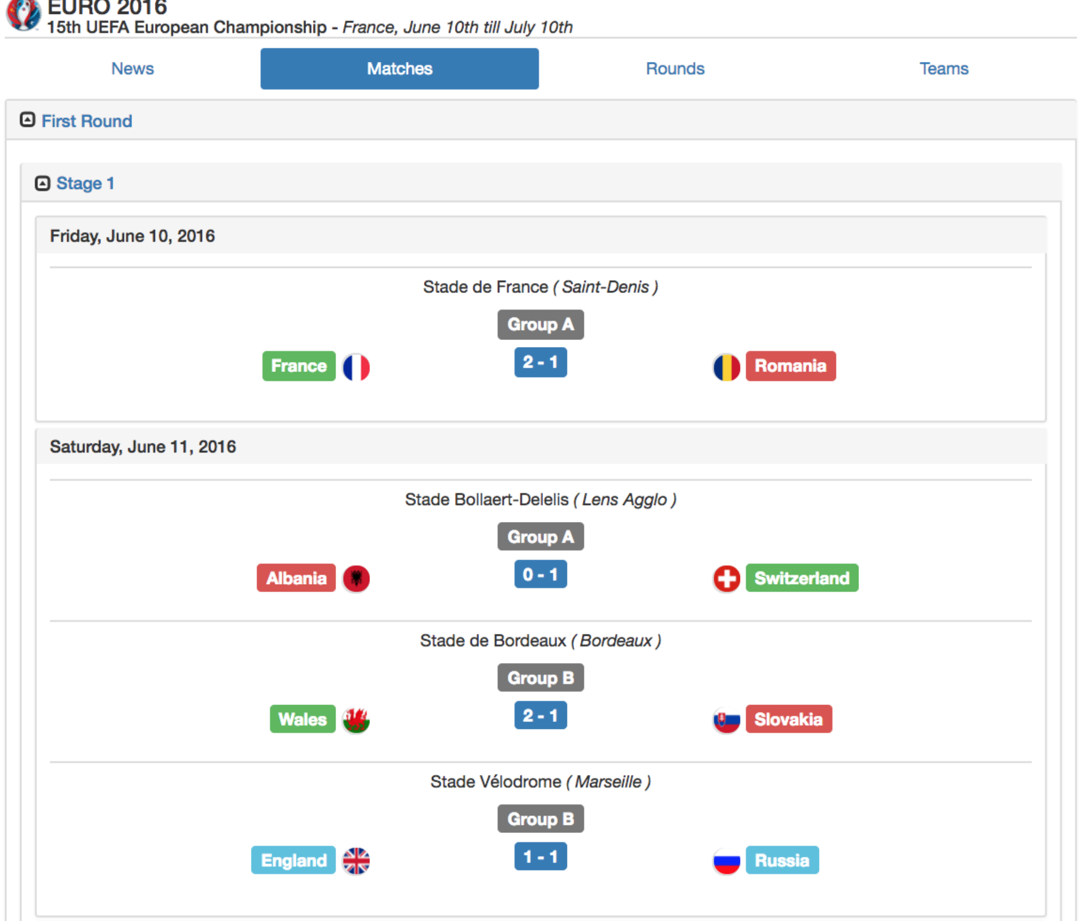
Finally, we presented in this chapter some limitations of our approach in addition to the conclusion, internal, construct and external threats to validity.



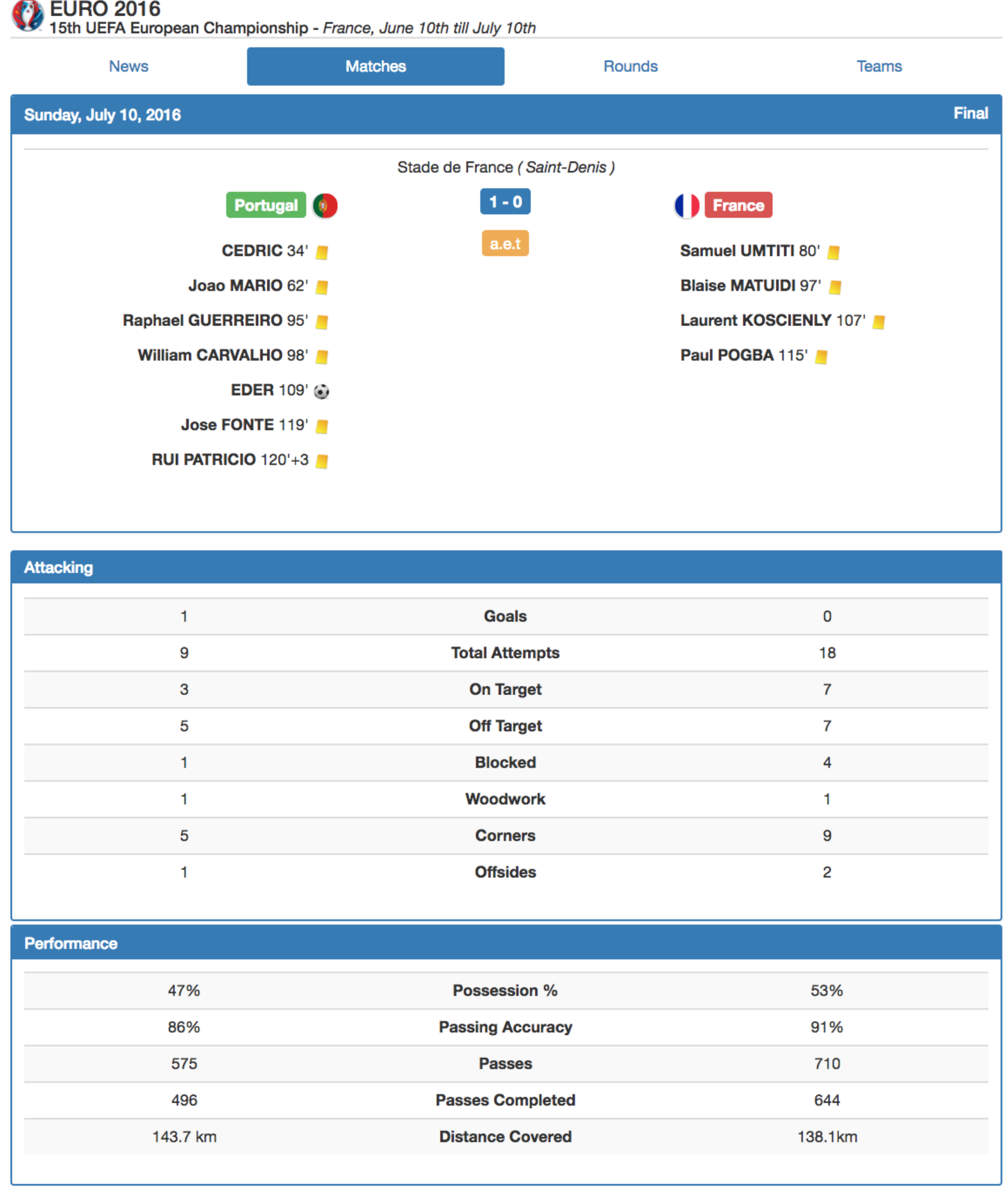
(a) Matches interface of product p_7

(b) Match details interface of product p_7

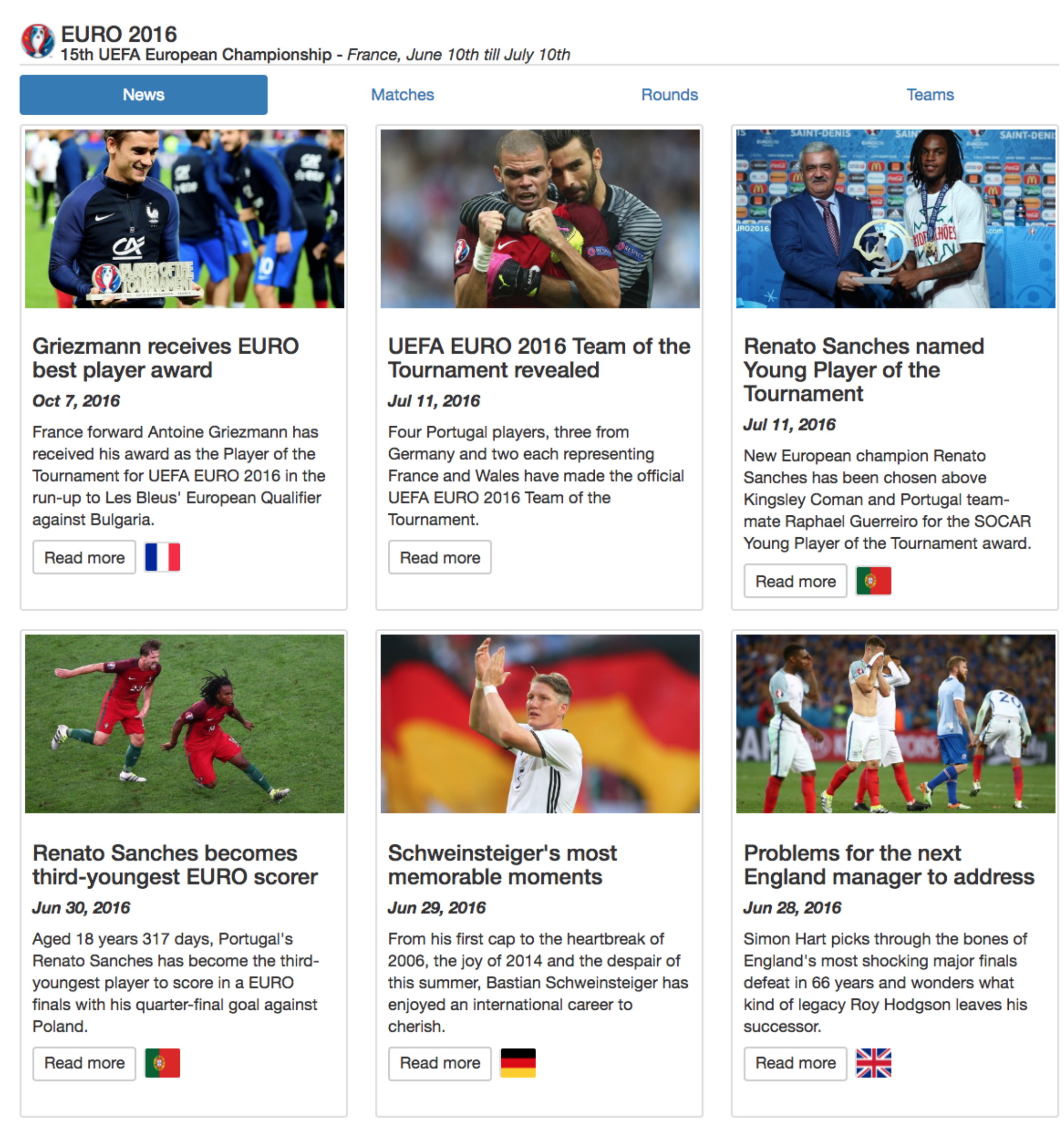
Figure 9.1: Case study graphical interfaces of product p_7



(a) Matches interface of product p_8

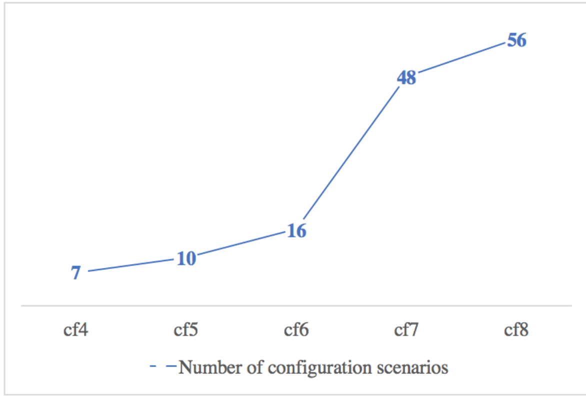


(b) Match details interface of product p_8

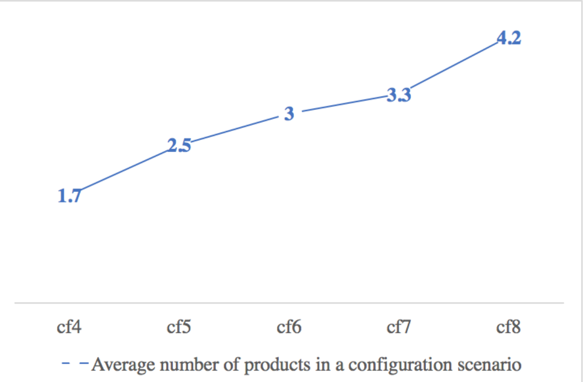


(c) News interface of product p_8

Figure 9.2: Case study graphical interfaces of product p_8

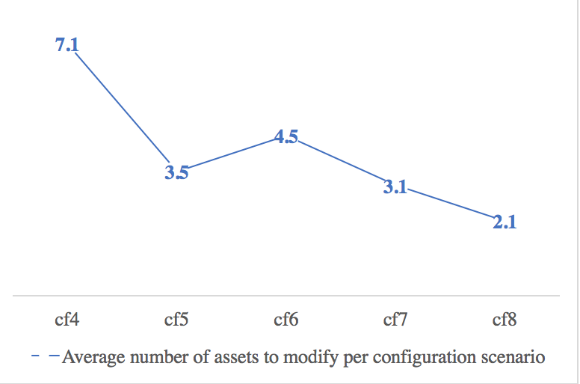


(a) Number of configuration scenarios per configuration

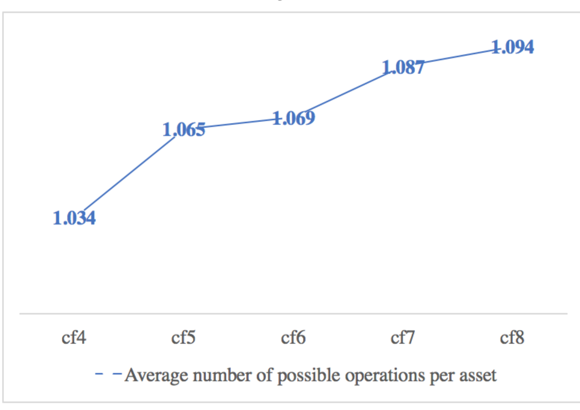


(b) Average number of products in a configuration scenario per configuration

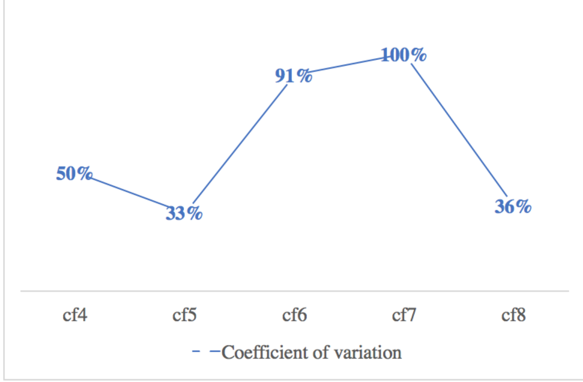
(c) Number of required existing assets per configuration



(d) Average number of assets to modify per configuration scenario per configuration



(e) Average number of operations per asset per configuration



(f) Coefficient of variation of configuration scenarios cost per configuration

Figure 9.3: Case study indicators

Part IV

Conclusion and Perspectives

CHAPTER 10

CONCLUSION AND PERSPECTIVES

Contents

10.1 Conclusion **136**
10.2 Perspectives **138**

10.1 Conclusion

In this dissertation, we proposed an approach to support the derivation of unplanned product variants in a software product line context using clone-and-own practice. The proposed approach allows the integration of existing product variants into a software product line. Moreover, it supports the derivation of new product variants using clone-and-own, by proposing the possible scenarios and operations to achieve the derivation. This support is strengthened by a cost estimation of the possible scenarios and operations and a constraints system guiding software engineers to perform the derivation based on their own preferences. Furthermore, the proposed approach enables the integration of the derived products into the software product line, to permit their reuse in future derivation.

Our approach addresses software families developed in a feature-oriented development context using clone-and-own, where the functionalities implemented in product variants are expressed as business-level features. To allow a systematic reuse of product variants, our approach offers an automated integration of existing product variants into a software product line. This process is accompanied with an automated and controlled construction of a feature model called restrictive FM, to enable the configuration and automated derivation of the existing product variants.

Since arising customer requirements and technology changes necessitate the development of new products, our approach fulfills this need by allowing the derivation of new product variants from the constructed product line. Therefore, our approach allows the definition of new features during the configuration of the restrictive FM, whenever the features required for a new product consist of a set of features realized by an existing product and some new features not implemented yet. However, if the required features are not realized by a single product, the restrictive FM does not permit their selection, regardless if they contain new features or not. For this purpose, another feature model is generated, as a constraint-free version of the restrictive FM, where all features except root feature are optional. Indeed, this feature model called free FM allows the configuration of new products, by selecting a set of features that is not realized by a single existing product, with the possibility of defining new features throughout the configuration as well.

In order to reuse artifacts from existing products to derive a new one, the products implementing the required existing features, and respectively the assets that accomplished their implementation have to be identified. Therefore, we defined an automated mechanism to identify mappings between the product line artifacts. Mappings are called correlations, and correlations are established between features and assets on the one side, and between feature and asset instances on the other side. Asset instances correspond to the versions of the assets.

Relying on the identified correlations, and according to the features selected in a configuration, our approach automatically determines the possible configuration scenarios and respectively operations to perform, in order to achieve the derivation of a new product. Operations correspond to actions to take over asset instances, in order to construct the necessary asset instances for the derivation. Such actions can be a clone of an existing instance, with the possibility of adding or removing implementation fragments corresponding respectively to required or unrequired features. A configuration scenario is a set of products that constitutes a source to achieve the derivation of a new product. Hence, to support software engineers in achieving the derivation,

our approach proposes its possible configuration scenarios and operations.

We strengthen our support by generating a constraints system for each configuration, in order to facilitate the selection of a configuration scenario and operations. This constraints system is constructed as a feature model that we call derivation FM. The features of the derivation FM are the configuration scenarios, their products, their operations and their corresponding assets and asset instances, while the constraints correspond to the dependencies between them. Hence, the generated derivation FM allows software engineers to select the favorable configuration scenario and operations based on their own preferences i.e. by selecting the configuration scenario that requires the least number of operations imposing a construction of a new asset instance, or the configuration scenario composed of the products that they are most familiar, or the operations involving the asset instances that they worked on.

Since the number of possible configuration scenarios and operations to achieve a product derivation can be elevated, we defined a cost-estimation function to estimate the cost in terms of efforts and development time that might be required to perform an operation. Respectively, the operations estimated cost allows to estimate the cost of configuration scenarios. Thus, we provide software engineers with the cost estimation as an additional argument that they can rely on to select their favorable configuration scenario and operations.

After the derivation of a new product, to allow its reuse in future derivations, our approach permits its automated integration in the product line. Hence, our approach enables the evolution of the product line, by integrating new products and respectively, by automatically updating the correlations and auto re-generating the restrictive and free FMs.

We implemented our approach by developing a framework called SUCCEED that stands for SUpporting Clone-and-own with Cost-EstimatEd Derivation. SUCCEED provides graphical user interfaces, which allow to migrate a set of product variants into a product line, support the configuration and derivation of a new variant, and the enrichment of the product line with new products.

We validated our approach based on experiments made on a real case study consisting of 8 product variants, by performing an incremental derivation and integration of 5 variants into a product line composed initially of 3 variants. The results revealed the importance of our approach in providing the necessary support to software engineers to accomplish the derivation of new products. Experiments showed that our approach supports and simplifies the derivation by automating several time-consuming and tedious tasks, by facilitating the choice of the favorable configuration scenario and operations and by providing an environment allowing software engineers to make their choices based on their own preferences and based on the proposed cost estimations.

In our approach, we consider software engineers as the main decision-makers. Hence, we support them with valuable elements to derive new products by themselves, without imposing on them a specific or automated derivation solution. By this, we preserve the “own” strength point of the clone-and-own approach, where software engineers own their code since they know the source of the clones and how it was constructed.

10.2 Perspectives

In our perspectives, we aim to validate our approach on more sophisticated systems of different architectures such as industrial case studies. Further, we aim to measure the effectiveness of our approach in terms of efforts and time saving, when compared to the classic clone-and-own approach. Such experiments should allow to determine valuable indicators that show the variation between the time spent by software engineers to manually accomplish derivation tasks compared to the automation that our approach provides of those tasks.

We consider our approach as complementary to several related works. Therefore, as future work, we are interested for example in integrating the techniques used by [FLLHE14] and [MZB+15b], to identify mappings between features and assets of monoglot product variants, but preserving the clone-and-own decisions on behalf of software engineers.

Finally, we give interest in integrating our approach in a wider development context, in which we take into consideration organizational factors and international distortions when defining our cost estimation functions, such as coordination between development team members, development phases, criteria selection and prioritization, product release and customer feedback [BBS11, MBF12, BWW+18].

LIST OF ABBREVIATIONS

C&O *Clone-and-Own*

CT *Correlation Type*

FCA *Formal Concept Analysis*

FM *Feature Model*

FODA *Feature-Oriented Domain Analysis*

LPL *Ligne de Produits Logiciels*

PV *Product Variant*

SPL *Software Product Line*

SPLE *Software Product Line Engineering*

SUCCEED *SUpporting Clone-and-own with Cost-EstimatEd Derivation*

VCS *Version Control System*

LIST OF FIGURES

1.1 Main interfaces of the running example variants 5

2.1 Software Product Line Activities [Nor02] 19
2.2 Software Product Line Engineering Framework [PBV05] 20
2.3 Costs for developing n kinds of systems as single systems compared to product line engineering [PBV05] 22
2.4 Time to market with and without product line engineering [PBV05] 23
2.5 Negative and positive variability [VG07] 24
2.6 A sample feature model and a configuration [BSRC10] 26

3.1 Relevant activities during extractive SPL adoption for leveraging artefact variants [Mar16] 33
3.2 Word clouds visualization functionality for feature naming [Mar16] 35
3.3 The *ECCO* workflow [FLLHE15] 42
3.4 The *ECCO* version control system workflow [Lin16] 43
3.5 Challenges and objectives 50
3.6 Approach overview 53

4.1 Migration process 57
4.2 Running example $\mathcal{SPL}$ global FM 60
4.3 $\mathcal{SPL}$ model diagram 63
4.4 Examples of feature to asset correlation 65
4.5 Correlations model diagram 66

5.1 Derivation process 76
5.2 An $\mathcal{SPL}$ example 77
5.3 Running example Restrictive and Free FMs 78
5.4 Possible scenarios to achieve a configuration 80

7.1 $\mathcal{SPL}$ evolution process. 98
7.2 Running example cf_4 derivation FM 101

8.1 An example of a configuration in restrictive mode 116
8.2 An example of a configuration in free mode 116
8.3 Running example derivation process in SUCCEED 118

9.1 Case study graphical interfaces of product p_7 127
9.2 Case study graphical interfaces of product p_8 130
9.3 Case study indicators 131

LIST OF TABLES

1.1 Running example product variants with their corresponding features 3
1.2 Running example product variants with an excerpt of their corresponding assets 4

2.1 A configuration space corresponding to the FM of Figure 2.6 27

3.1 A comparison of the key characteristics of the main related work tools 49

4.1 Running example product variants with their implementation files and feature models 59
4.2 Running example relationships between assets, asset instances and product variants 63
4.3 Correlations between features and assets of the running example 66
4.4 Correlations between features and asset instances of the running example . . . 67
4.5 Product variants with their corresponding features 70
4.6 Product variants with their corresponding asset instances 70
4.7 Correlations between features and assets 70

5.1 Possible configuration scenarios of cf_4 with their possible operations 86

6.1 Running example asset instance correlated features 91
6.2 Running example feature correlated assets 92
6.3 Running example instances of assets 92
6.4 Running example features correlated asset instances 93
6.5 Correlation degrees between features and asset instances of the running example 93
6.6 Configuration cf_4 actions estimated cost 94
6.7 Configuration cf_4 operations estimated cost 94
6.8 Configuration cf_4 configuration scenarios estimated cost 95

7.1 Product p_4 asset instances 103
7.2 Added correlations after integration of product p_4 105
7.3 Updated running example asset instance correlated features after integration of product p_4 106
7.4 Updated running example instances of assets after integration of product p_4 . . 106
7.5 Updated running example features correlated asset instances after integration of product p_4 106
7.6 Updated correlation degrees between features and asset instances of the running example after integrating product p_4 107

9.1 Metrics of 5 sequential configurations to derive new PVs 122
9.2 Main features of case study product variants 123

LIST OF ALGORITHMS

1 Assets extraction and products definition 114
2 Correlations identification 115

TABLE OF OBJECTIVES

O.1 Reuse in feature-oriented software development . . . 15
O.2 Adopting clone-and-own as a starting point . . . 17
O.3 Relying on software product line as a sustainable solution . . . 23
O.4 Modeling variability through a feature model . . . 26
O.5 Combining clone-and-own and software product line . . . 29
O.6 Extractive migration approach . . . 32
O.7 Business-level feature identification . . . 36
O.8 Global mapping between features and assets . . . 37
O.9 Language-independent mappings identification . . . 38
O.10 Capturing features interactions . . . 39
O.11 Smooth migration in the accurate moment . . . 41
O.12 Supporting derivation with the possible scenarios . . . 44
O.13 Cost-estimated derivation . . . 44
O.14 Software engineers as decision makers . . . 44
O.15 Reactive SPL evolution . . . 46
O.16 Incremental traceability between features and assets . . . 46
O.17 Allow a complete reuse . . . 47
O.18 Managing feature model evolution . . . 47

TABLE OF DEFINITIONS AND PROPERTIES

4.2.1 Definition (Feature) . . . 58
4.2.2 Definition (Feature Model) . . . 58
4.2.3 Definition (Restrictive FM) . . . 59
4.2.4 Definition (Asset Instance) . . . 60
4.2.5 Definition (Asset) . . . 61
4.2.6 Definition (Product) . . . 61
4.2.7 Definition (Product implements features) . . . 61
4.2.8 Definition (Software Product Line $\mathcal{SPL}$) . . . 61
4.2.9 Definition (Artifact) . . . 62
4.2.10 Definition (Product employs assets and exploits asset instances) . . . 62
4.2.11 Definition (Feature implemented by products) . . . 62
4.2.12 Definition (Asset employed by products) . . . 62
4.2.13 Definition (Asset instance exploited by products) . . . 62
4.2.1 Property ($\mathcal{SPL}$ facts) . . . 62
4.2.2 Property (Asset instance uniqueness in product) . . . 62
4.3.1 Definition (Correlation) . . . 64
4.3.2 Definition (Feature to Asset Correlation) . . . 64
4.3.3 Definition (Correlation Type) . . . 64
4.3.4 Definition (Feature to Asset Instance Correlation) . . . 66
4.3.5 Definition (Asset correlated features) . . . 67
4.3.6 Definition (Asset instance correlated features) . . . 67
4.3.7 Definition (Feature correlated assets) . . . 67
4.3.8 Definition (Feature correlated asset instances) . . . 67
4.4.1 Property (Complete $\mathcal{SPL}$) . . . 69

5.2.1 Definition (Configuration) . . . 75
5.2.1 Property (Product contributes in configuration) . . . 75
5.2.2 Property (Product realizes a configuration) . . . 75
5.2.3 Property (Product covers a configuration) . . . 77
5.2.2 Definition (Free FM) . . . 78
5.4.1 Definition (Configuration scenario) . . . 79
5.4.1 Property (Products of a configuration scenario) . . . 79
5.5.1 Definition (Required assets for derivation) . . . 83
5.5.2 Definition (Action) . . . 84
5.5.3 Definition (Operation) . . . 84

6.2.1 Definition (Action type weight) . . . 90
6.2.2 Definition (Correlation degree) . . . 91
6.2.3 Definition (Action cost) . . . 92

6.2.4 Definition (Operation cost) 94
6.2.5 Definition (Configuration Scenario cost) 95

7.2.1 Definition (Derivation FM) 99
7.2.2 Definition (Derivation scenario) 99

TABLE OF EXAMPLES

1 Product variants implementation files and feature models . . . 58
2 Applying the FAMILIAR merge operation to construct the $\mathcal{SPL}$ FM . . . 60
3 Running example extracted assets and instances . . . 62
4 Running example feature to asset correlations . . . 64
5 Running example feature to asset instance correlations . . . 67
6 Example of uncorrelated asset if R5 violated . . . 70
7 Running example Restrictive and Free FMs . . . 78
8 Running example configuration of a new product . . . 82
9 Some actions on a configuration from running example . . . 84
10 Some operations on a configuration from running example . . . 85
11 Asset instance correlated features from running example . . . 91
12 Feature correlated assets from running example . . . 92
13 Instances of assets from running example . . . 92
14 Feature correlated asset instances from running example . . . 92
15 Correlation degrees from running example . . . 93
16 Actions cost from running example . . . 93
17 Operations cost from running example . . . 94
18 Configuration scenario cost from running example . . . 95
19 Configuration scenarios having same cost . . . 96
20 Running example derivation FM . . . 100
21 Selection based on numbers and indicators . . . 100
22 Selection based on developer preferences . . . 101
23 Selection based on cost estimation . . . 102
24 Integration of product p_4 of the running example . . . 103
25 FAMILIAR merge operation to re-generate $\mathcal{SPL}$ restrictive FM . . . 104
26 Free FM eventual re-generation . . . 104
27 Running example correlations update . . . 105
28 Running example correlation indicators update . . . 105

LISTINGS

4.1 Running example PVs FMs in FAMILIAR language 60
4.2 FAMILIAR merge operation over PVs FMs 60
4.3 $\mathcal{SPL}$ FM generated from FAMILIAR merge operation 60
7.1 Restrictive FM in FAMILIAR language before merge 104
7.2 FM of product p_4 104
7.3 FAMILIAR merge operation over $\mathcal{SPL}$ restrictive FM and FM of p_4 104
7.4 Re-generated restrictive FM from FAMILIAR merge operation 104

BIBLIOGRAPHY

[ABKS16] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, 2016.

[Ach11] Mathieu Acher. *Managing, multiple feature models: foundations, languages and applications*. PhD thesis, Nice, 2011.

[ACLF13] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Familiar: A domain-specific language for large scale management of feature models. In *Science of Computer Programming*, volume 78, pages 657–681. Elsevier, June 2013.

[ACR09] Hugo Arboleda, Rubby Casallas, and Jean-Claude Royer. Dealing with fine-grained configurations in model-driven SPLs. In *SPLC*, pages 1–10, 2009.

[AG06] Giuliano Antoniol and Yann-Gael Gueheneuc. Feature identification: An epidemiological metaphor. *IEEE Trans. Softw. Eng.*, 32(9):627–641, September 2006.

[AGM+06] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos Lucena. Refactoring product lines. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, pages 201–210, New York, NY, USA, 2006. ACM.

[AHC+12] Mathieu Acher, Patrick Heymans, Philippe Collet, Clément Quinton, Philippe Lahire, and Philippe Merle. Feature model differences. In *Advanced Information Systems Engineering*, pages 629–645. Springer, 2012.

[AK08] Samuel A Ajila and Ali B Kaba. Evolution support mechanisms for software product line process. *Journal of Systems and Software*, 81(10):1784–1801, 2008.

[AK09] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.

[ALHL+17] Wesley KG Assunção, Roberto E Lopez-Herrejon, Lukas Linsbauer, Silvia R Vergilio, and Alexander Egyed. Reengineering legacy applications into software product lines: a systematic mapping. *Empirical Software Engineering*, 22(6):2972–3016, 2017.

[AM14] R AL-Msie'Deen. *Reverse Engineering Feature Models from Software Variants to Build Software Product Lines*. PhD thesis, PhD thesis, University of Montpellier, 2014.

[AmSH+13] Ra'fat Al-msie'deen, Abdelhak-Djamel Seriai, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Mining features from the object-oriented source code of software variants by combining lexical and structural similarity. In *Proceedings of the 2013 IEEE 14th International Conference on Information Reuse & Integration (IRI)*, pages 586–593, San Francisco, CA, 2013.

[ANCM12] Vallabh Anwikar, Ravindra Naik, Adnan Contractor, and Hemanth Makkapati. Domain-driven technique for functionality identification in source code. *SIGSOFT Softw. Eng. Notes*, 37(3):1–8, May 2012.

[AR13] Hugo Arboleda and Jean-Claude Royer. *Model-Driven and Software Product Line Engineering*. John Wiley & Sons, 2013.

[Ara89] G. Arango. Domain analysis: from art form to engineering discipline. In *Proceedings of the 5th international workshop on Software specification and design - IWSSD '89*, volume 14, pages 152–159, New York, New York, USA, April 1989. ACM Press.

[AV14] Wesley Klewerton Guez Assunção and Silvia Regina Vergilio. Feature location for software product line migration: a mapping study. In *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools-Volume 2*, pages 52–59. ACM, 2014.

[Bat05] Don Batory. Feature models, grammars, and propositional formulas. In *SPLC*, volume 3714, pages 7–20. Springer, 2005.

[BBS11] Jan Bosch and Petra M. Bosch-Sijtsema. Introducing agile customer-centered development in a legacy software product line. *Softw. Pract. Exper.*, 41(8):871–882, July 2011.

[BEG+11] Ebrahim Bagheri, Faezeh Ensan, Dragan Gasevic, Marko Boskovic, et al. Modular feature models: Representation and configuration. *Journal of Research and Practice in Information Technology*, 43(2):109, 2011.

[Bel08] Radim Belohlavek. Introduction to formal concept analysis. *Palacky University, Department of Computer Science, Olomouc*, 2008.

[BFK+99] Joachim Bayer, Oliver Flege, Peter Knauber, Roland Laqua, Dirk Muthig, Klaus Schmid, Tanya Widen, and Jean-Marc DeBaud. Pulse: A methodology to develop software product lines. In *Proceedings of the 1999 Symposium on Software Reusability*, SSR '99, pages 122–131, New York, NY, USA, 1999. ACM.

[BL07] Hongyu Pei Breivold and Magnus Larsson. Component-based and service-oriented software engineering: Key concepts and principles. In *Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on*, pages 13–20. IEEE, 2007.

[BLnC16] Manuel Ballarin, Raúl Lapeña, and Carlos Cetina. Leveraging feature location to extract the clone-and-own relationships of a family of software products. In *Proceedings of the 15th International Conference on Software Reuse: Bridging with Social-Awareness - Volume 9679*, ICSR 2016, pages 215–230, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[BLR+15] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. What is a feature?: A qualitative study of features in industrial software product lines. In *Proceedings of the 19th International Conference on Software Product Line*, SPLC '15, pages 16–25, New York, NY, USA, 2015. ACM.

[Bos07] Jan Bosch. Software product families: towards compositionality. *Fundamental Approaches to Software Engineering*, pages 1–10, 2007.

[Bos10] Jan Bosch. Toward compositional software product lines. *IEEE software*, 27(3):29–34, 2010.

[BP14] Goetz Botterweck and Andreas Pleuss. *Evolution of Software Product Lines*, pages 265–295. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[BPD+10] Goetz Botterweck, Andreas Pleuss, Deepak Dhungana, Andreas Polzer, and Stefan Kowalewski. Evofm: Feature-driven planning of product-line evolution. In *Proceedings of the 2010 ICSE Workshop on Product Line Approaches in Software Engineering*, PLEASE '10, pages 24–31, New York, NY, USA, 2010. ACM.

[BPPK09] Goetz Botterweck, Andreas Pleuss, Andreas Polzer, and Stefan Kowalewski. Towards feature-driven planning of product-line evolution. In *Proceedings of the First International Workshop on Feature-Oriented Software Development*, FOSD '09, pages 109–116, New York, NY, USA, 2009. ACM.

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[BWW+18] Deepika Badampudi, Krzysztof Wnuk, Claes Wohlin, Ulrik Franke, Darja Smite, and Antonio Cicchetti. A decision-making process-line for selection of software asset origins and components. *Journal of Systems and Software*, 135:88–104, 2018.

[CA05] Krzysztof Czarnecki and Michał Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *International conference on generative programming and component engineering*, pages 422–437. Springer, 2005.

[CAK+05] Krzysztof Czarnecki, Michal Antkiewicz, Chang Hwan Peter Kim, Sean Lau, and Krzysztof Pietroszek. Model-driven software product lines. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 126–127. ACM, 2005.

[CBK+13] Rafael Capilla, Jan Bosch, Kyo-Chul Kang, et al. Systems and software variability management. *Concepts Tools and Experiences*, 2013.

[CHE04] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration using feature models. In *SPLC*, volume 3154, pages 266–283. Springer, 2004.

[CN01] Paul Clements and Linda M Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, 2001.

[CS14] Scott Chacon and Ben Straub. *Pro Git, Second Edition*. Apress, 2014.

[Dav87] Stanley M. Davis. *Future Perfect*. Addison-Wesley, Boston, Massachusetts, 1987.

[DDF+90] Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391, 1990.

[DDH+13] Jean-Marc Davril, Edouard Delfosse, Negar Hariri, Mathieu Acher, Jane Cleland-Huang, and Patrick Heymans. Feature model extraction from large collections of informal product descriptions. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 290–300. ACM, 2013.

[DRB+13] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An exploratory study of cloning in industrial software product lines. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 25–34. IEEE, 2013.

[DRGP13] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of software: Evolution and Process*, 25(1):53–95, 2013.

[DSB05] Sybren Deelstra, Marco Sinnema, and Jan Bosch. Product derivation in software product families: a case study. *Journal of Systems and Software*, 74(2):173–194, January 2005.

[FF98] B. Faltings and E.C. Freuder. Configuration [Guest Editor's Introduction]. *IEEE Intelligent Systems and their Applications*, 13(4):32–33, July 1998.

[FLLHE14] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. Enhancing clone-and-own with systematic reuse for developing software variants. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 391–400. IEEE, 2014.

[FLLHE15] Stefan Fischer, Lukas Linsbauer, Roberto E Lopez-Herrejon, and Alexander Egyed. The ecco tool: Extraction and composition for clone-and-own. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 665–668. IEEE Press, 2015.

[FV03] David Faust and Chris Verhoef. Software product line migration and deployment. *Software: Practice and Experience*, 33(10):933–955, 2003.

[GH04] Hassan Gomaa and Mohamed Hussein. Software reconfiguration patterns for dynamic evolution of software architectures. In *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*, pages 79–88. IEEE, 2004.

[Gro17] Gartner Group. Gartner Says Worldwide IT Spending Forecast to Grow 2.4 Percent in 2017. https://www.gartner.com/newsroom/id/3759763, 2017. [Online; accessed 27-December-2017].

[Her85] American Heritage. *The American Heritage Dictionary*. 1985.

[Hul03] Anette Hulth. Improved automatic keyword extraction given more linguistic knowledge. In *Proceedings of the 2003 conference on Empirical methods in natural language processing*, pages 216–223. Association for Computational Linguistics, 2003.

[HVLG12] Wolfgang Heider, Michael Vierhauser, Daniela Lettner, and Paul Grunbacher. A case study on the evolution of a component-based product line. In *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*, pages 1–10. IEEE, 2012.

[HWK06] L. Hotz, K. Wolter, and T. Krebs. *Configuration in Industrial Product Families: The ConIPF Methodology*. IOS Press, Inc., 2006.

[IRBW16] Nili Itzik, Iris Reinhartz-Berger, and Yair Wand. Variability analysis of requirements: Considering behavioral differences and reflecting stakeholders' perspectives. *IEEE Transactions on Software Engineering*, 42(7):687–706, 2016.

[JBAC15] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. Maintaining feature traceability with embedded annotations. In *Proceedings of the 19th International Conference on Software Product Line*, pages 61–70. ACM, 2015.

[JLL05] He Jifeng, Xiaoshan Li, and Zhiming Liu. Component-based software engineering. *Lecture notes in computer science*, 3722:70, 2005.

[Jun08] Ulrich Junker. Preference-based problem solving for constraint programming. In *Recent Advances in Constraints*, pages 109–126. 2008.

[KA13] Christian Kästner and Sven Apel. Feature-oriented software development. In *Generative and Transformational Techniques in Software Engineering IV*, pages 346–382. Springer, 2013.

[KCH+90] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[KDO14] Christian Kästner, Alexander Dreiling, and Klaus Ostermann. Variability mining: Consistent semi-automatic detection of product-line features. *IEEE Transactions on Software Engineering*, 40(1):67–82, 2014.

[KLD02] Kyo C Kang, Jaejoon Lee, and Patrick Donohoe. Feature-oriented product line engineering. *IEEE software*, 19(4):58–65, 2002.

[KNGDNK+16] Ganesh Khandu Narwane, José Angel Galindo Duarte, Shankara Narayanan Krishna, David Benavides, Jean-Vivien Millo, and S Ramesh. Traceability analyses between features and assets in software product lines. *Entropy*, July 2016.

[Kru92] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.

[Kru01] CharlesW Krueger. Easing the transition to software mass customization. In *International Workshop on Software Product-Family Engineering*, pages 282–293. Springer, 2001.

[LBC16] Raúl Lapeña, Manuel Ballarin, and Carlos Cetina. Towards clone-and-own support: locating relevant methods in legacy products. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 194–203. ACM, 2016.

[LBL06] Jia Liu, Don Batory, and Christian Lengauer. Feature oriented refactoring of legacy applications. In *Proceedings of the 28th international conference on Software engineering*, pages 112–121. ACM, 2006.

[LELH16] Lukas Linsbauer, Alexander Egyed, and Roberto Erick Lopez-Herrejon. A variability aware configuration management and revision control platform. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 803–806. ACM, 2016.

[LFL98] Thomas K Landauer, Peter W Foltz, and Darrell Laham. An introduction to latent semantic analysis. *Discourse processes*, 25(2-3):259–284, 1998.

[LHLE15] Roberto E Lopez-Herrejon, Lukas Linsbauer, and Alexander Egyed. A systematic mapping study of search-based software engineering for software product lines. *Information and software technology*, 61:33–51, 2015.

[Lin16] Lukas Linsbauer. Feature-oriented and distributed version control system, 2016.

[LKL02] Kwanwoo Lee, Kyo C Kang, and Jaejoon Lee. Concepts and guidelines of feature modeling for product line software engineering. In *International Conference on Software Reuse*, pages 62–77. Springer, 2002.

[LLHE16] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. Variability extraction and modeling for product variants. *Software & Systems Modeling*, Jan 2016.

[LnFPC16] Raúl Lapeña, Jaime Font, Francisca Pérez, and Carlos Cetina. Improving feature location by transforming the query from natural language into requirements. In *Proceedings of the 20th International Systems and Software Product Line Conference*, SPLC '16, pages 362–369, New York, NY, USA, 2016. ACM.

[LnFPC17] Raúl Lapeña, Jaime Font, Óscar Pastor, and Carlos Cetina. Analyzing the impact of natural language processing over feature location in models. *SIGPLAN Not.*, 52(12):63–76, October 2017.

[LSR07] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[LYSL07] Yi-yuan Li, Jian-wei Yin, Dong-cai Shi, and Yin Li. Feature configuration modeling and problem solving for software product line. In *Computer and Computational Sciences, 2007. IMSCCS 2007. Second International Multi-Symposiums on*, pages 531–536. IEEE, 2007.

[MAGDC+16] David Méndez-Acuña, José Angel Galindo Duarte, Benoit Combemale, Arnaud Blouin, Benoit Baudry, and Gurvan Le Guernic. Reverse-engineering reusable language modules from legacy domain-specific languages. In *International Conference on Software Reuse*, Proceedings of the International Conference on Software Reuse, Limassol, Cyprus, June 2016.

[Mar16] Jabier Martinez. *Mining software artefact variants for product line migration and analysis*. Theses, Université Pierre et Marie Curie - Paris VI, October 2016.

[MBF12] Ana Magazinius, Sofia Börjesson, and Robert Feldt. Investigating intentional distortions in software cost estimation–an exploratory study. *Journal of Systems and Software*, 85(8):1770–1781, 2012.

[ME08] Ralf Mitschke and Michael Eichberg. Supporting the evolution of software product lines. In *ECMDA Traceability Workshop (ECMDA-TW)*, pages 87–96, 2008.

[MP14] Andreas Metzger and Klaus Pohl. Software product line engineering and variability management: achievements and challenges. *Proceedings of the on Future of Software Engineering - FOSE 2014*, pages 70–84, 2014.

[MZB+15a] Jabier Martinez, Tewfik Ziadi, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Automating the extraction of model-based software product lines from model variants (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 396–406. IEEE, 2015.

[MZB+15b] Jabier Martinez, Tewfik Ziadi, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Bottom-up adoption of software product lines: a generic and extensible approach. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, pages 101–110, 2015.

[MZB+16] Jabier Martinez, Tewfik Ziadi, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Name suggestions during feature identification: the variclouds approach. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 119–123. ACM, 2016.

[MZB+17] Jabier Martinez, Tewfik Ziadi, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Bottom-up technologies for reuse: Automated extractive adoption of software product lines. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 67–70. IEEE Press, 2017.

[MZKLT14] Jabier Martinez, Tewfik Ziadi, Jacques Klein, and Yves Le Traon. Identifying and visualising commonality and variability in model variants. In *ECMFA 2014 European Conference on Modelling Foundations and Applications*, 2014.

[MZM+14] Jabier Martinez, Tewfik Ziadi, Raul Mazo, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Feature relations graphs: A visualisation paradigm for feature constraints in software product lines. In *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*, pages 50–59. IEEE, 2014.

[NBA+15] L. Neves, P. Borba, V. Alves, L. Turnes, L. Teixeira, D. Sena, and U. Kulesza. Safe evolution templates for software product lines. *J. Syst. Softw.*, 106(C):42–58, August 2015.

[NCB+07] Linda Northrop, Paul Clements, Felix Bachmann, John Bergey, Gary Chastek, Sholom Cohen, Patrick Donohoe, Lawrence Jones, Robert Krut, Reed Little, et al. A framework for software product line practice, version 5.0. *SEI.–2007–http://www. sei. cmu. edu/productlines/index. html*, 2007.

[Nor02] Linda M Northrop. Sei's software product line tenets. *IEEE software*, 19(4):32–40, 2002.

[Par76] David Lorge Parnas. On the design and development of program families. *IEEE Transactions on software engineering*, (1):1–9, 1976.

[PBD+12] Andreas Pleuss, Goetz Botterweck, Deepak Dhungana, Andreas Polzer, and Stefan Kowalewski. Model-driven support for product line evolution on feature level. *J. Syst. Softw.*, 85(10):2261–2274, October 2012.

[PBV05] K. Pohl, G. Böckle, and F. Van Der Linden. Software Product Line Engineering. Foundations, Principles, and Techniques. *Uwplatt.Edu*, 49(12):467, 2005.

[RC12] Julia Rubin and Marsha Chechik. Combining related products into product lines. In *Fase*, volume 12, pages 285–300. Springer, 2012.

[RC13a] Julia Rubin and Marsha Chechik. A framework for managing cloned product variants. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1233–1236. IEEE Press, 2013.

[RC13b] Julia Rubin and Marsha Chechik. A survey of feature location techniques. In *Domain Engineering*, pages 29–58. Springer, 2013.

[RCC13] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. Managing cloned variants: a framework and experience. In *Proceedings of the 17th International Software Product Line Conference*, pages 101–110. ACM, 2013.

[RKBC12] Julia Rubin, Andrei Kirshin, Goetz Botterweck, and Marsha Chechik. Managing forked product variants. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 156–160. ACM, 2012.

[RSH+13] AL Ra'Fat, Abdelhak Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Hamzeh Eyal Salman. Feature location in a collection of software product variants using formal concept analysis. In *International Conference on Software Reuse*, pages 302–307. Springer, 2013.

[SB99] Mikael Svahnberg and Jan Bosch. Evolution in software product lines. *Software Maintenance*, 11(6):391–422, 1999.

[SBB+10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. *Software Product Lines: Going Beyond*, pages 77–91, 2010.

[Sch06] Douglas C Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.

[SE08] Klaus Schmid and Holger Eichelberger. A requirements-based taxonomy of software product line evolution. *Electronic Communications of the EASST*, 8, 2008.

[SHT06] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature diagrams: A survey and a formal semantics. In *Requirements Engineering, 14th IEEE international conference*, pages 139–148. IEEE, 2006.

[SHTB07] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Comput. Netw.*, 51(2):456–479, February 2007.

[SHU+13] Abdelhak-Djamel Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, Hamzeh Eyal-Salman, et al. Mining features from the object-oriented source code of a collection of software variants using formal concept analysis and latent semantic indexing. In *The 25th International Conference on Software Engineering and Knowledge Engineering*, page 8. Knowledge Systems Institute Graduate School, 2013.

[SRG11] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. A comparison of decision modeling approaches in product lines. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '11, pages 119–126, New York, NY, USA, 2011. ACM.

[SV02] Klaus Schmid and Martin Verlage. The economic impact of product line adoption and evolution. *IEEE software*, 19(4):50–57, 2002.

[TH03] Jean-Christophe Trigaux and Patrick Heymans. *Software Product Lines : State of the art*. PhD thesis, 2003.

[TKB+14] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014.

[VG07] Markus Voelter and Iris Groher. Product line implementation using aspect-oriented and model-driven software development. In *Proceedings of the 11th International Software Product Line Conference*, SPLC '07, pages 233–242, Washington, DC, USA, 2007. IEEE Computer Society.

[vGBS01] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *Proceedings of the Conference on Software Architecture*, pages 45–54, 2001.

[Wil96] Rudolf Wille. *Introduction to formal concept analysis*. Fachbereich Mathematik, Technische Hochschule Darmstadt, 1996.

[WL99] David M. Weiss and Chi Tau Robert Lai. *Software Product-line Engineering: A Family-based Software Development Process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[YPZ09] Yiming Yang, Xin Peng, and Wenyun Zhao. Domain feature model recovery from multiple applications using data access semantics and formal concept analysis. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pages 215–224. IEEE, 2009.

[ZFdSZ12] Tewfik Ziadi, Luz Frias, Marcos Aurelio Almeida da Silva, and Mikal Ziane. Feature identification from the source code of product variants. In *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*, CSMR '12, pages 417–422, Washington, DC, USA, 2012. IEEE Computer Society.

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

[ZPXZ12] Gang Zhang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. Cloning practices: Why developers clone and what can be changed. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 285–294. IEEE, 2012.