Universidade do Minho
Escola de Engenharia

Karam Ignaim

EvoSPL: An evolutionary approach for adopting software product lines in the automotive industry

fevereiro de 2021

EvoSPL: An evolutionary approach for adopting software product lines in the automotive industry

Karam Ignaim

UMinho | 2021

Universidade do Minho
Escola de Engenharia

Karam Ignaim

EvoSPL: An evolutionary approach for adopting software product lines in the automotive industry

PhD. Thesis
PhD. in Informatics

A project supervised by
Prof. João M. Fernandes,

Dr. André Ferreira

fevereiro de 2021

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.
Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.
I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho

Full name

Signature

Acknowledgements

"He who does not thank Allah does not thank people"

Prophet Mohammad (**PBUH**)

I wish to express my love and gratitude to my God for his grace and mercy, this work would never be possible without his support.

I would like to thank those who gave me the possibility to complete this thesis. First, I am deeply grateful to my supervisor **Prof. João Fernandes** for his support, encouragement and patience. He has guided me and encouraged me to carry on through these years and has contributed to this thesis with a major impact. I would like to thank my co-supervisor Dr. André Ferreira for his support to perform the industrial case study successfully.

I would like to thank Bosch Car Multimedia (the Bosch company at Braga) and our colleagues there. Special thanks to the Classical Sensor Software Team: João Santos, Hélder Vilas-Boas, and Joao Cardoso. I would like to show my gratitude to Jana Seidel (Project Manager) for her help and support.

I gratefully acknowledge the financial support from Erasmus and my home university (Al-Balqa' Appieled University - Jordan). I would like also to thank the Global Platform for Syrian Students for the financial support for the academic year 2019/2020. I am also very glad and thankful for having met Dr. Helena Barroco - Diplomatic Adviser to President Jorge Sampaio. I would like to express a special appreciation for her support.

'Muito obrigada pelas pessoas em Braga. Braga é uma cidade muito bonita'

I am also very grateful to all the staff in the University of Minho specifically, the staff of the library (Bibliotecas UM) - Campus de Gualtar, Departamento de Informática (Centero Algoritmi), Dr. César Analide (Escola de Engenharia), Carla Araújo (Técnico de Informática), Célia Afonso (Serviço Acadêmico), Helena Dias (Assistente técnica do Departamento de Informática), and

cleaning ladies. I would like to show my gratitude to the staff of the Complexo Desportivo dos Serviços de Acção Social da Universidade do Minho - Campus de Gualtar, who always gives me a family vibes.

I want to express my deep gratitude to Dr. Ali Shoker (research scientist) and Dr. Mathieu Acher, Univ Rennes, Inria, CNRS, IRISA, France (DiverSE team).

I have to say thank you to my family for keeping things going and for always showing how proud they are of me (my lovely sister Isra and brothers: Hani, Mohammad, and Yousef). Special thanks to my uncles: Prof. Kamal, JamaL, Prof. Natheer, and Dr. Mohammad Ahadidi, as well as to Dr. Aklas Alhadidi, Sajeda Ajo, Eng. Suha daher, and Ala Alhadidi.

A warm word for my lovely friends particularly, Amal Awamleh, Malak Alqurabi, Amani Shehadi, Amani Shoker, Fadwa Mhamdi, Rana Jour and Esra Jaradat, who have been the light of my life for the last years and who have given me the extra strength and motivation to get things done. Also, I would like to thank my friends, Francisco Costa, Diogo Tomé, Abbass Rjoub, Alsharawi Salem, Aniis Medane, André Morim, Samer Hamati, Bassam Jmail, Bassem Kheireddine and Rashed Esam.

Finally, the last word goes to my soul mate, who will appear one day in my life.

Dedication

بسم هلا الرحمن الرحيم

اقْرَأْ بِاسْمِ رَبِّكَ الَّذِي خَلَقَ - خَلَقَ الْإِنسَانَ مِنْ عَلَقٍ - اقْرَأْ وَرَبُّكَ الْأَكْرَمُ - الَّذِي عَلَّمَ بِالْقَلَمِ - عَلَّمَ الْإِنسَانَ مَا لَمْ يَعْلَمْ

"Read! In the Name of your Lord, Who has created (all that exists), Has created man from a clot. Read! And your Lord is the Most Generous, Who has taught (the writing) by the pen. Has taught man that which he knew not," [Al-'Alaq 96: 1-5]. Quran

This thesis is dedicated to my parents for their endless love, support, and encouragement. I would like to dedicate this work to the spirit of my father, who told me go and discover the world and I support you.

This thesis also is dedicated to my first country Jordan and my second country Portugal.

EvoSPL: Uma abordagem evolutiva para a adoção de linhas de produtos de software na indústria automotiva

Resumo

Algumas empresas desenvolvem uma família de produtos, criando a oportunidade de reutilizar e diferenciar os produtos da família. Na prática, o desenvolvimento e a evolução de tais produtos são normalmente realizados de maneira ad-hoc e as mudanças estão espalhadas por todos os artefactos de uma família. Porém, no longo prazo, tais abordagens ad-hoc apresentam grandes desvantagens para a manutenção e a evolução. Assim, é necessária uma reutilização sistemática. As linhas de produtos de software (SPLs) são amplamente adotadas pela indústria como uma ideia chave para reutilização sistemática. Para migrar os produtos existentes para uma SPL, especialmente na indústria automóvel, os profissionais precisam de uma abordagem completa.

Nesta tese, apresentamos uma abordagem evolutiva, denominada EvoSPL, para gerir a evolução de SPLs. Além disso, a abordagem EvoSPL adota um processo sistemático de reengenharia que é composto por três fases principais: engenharia reversa, engenharia direta e mapeamento. A fase de engenharia reversa captura as partes comuns e variáveis dos produtos existentes no Feature Model (FM), que representa uma SPL no domínio automóvel. A fase de engenharia direta inicia os produtos restantes de uma família (que não foram usados na fase de engenharia reversa) na SPL e, em seguida, trata da sua evolução sempre que um cliente solicita um novo produto individual. A fase de mapeamento específica que fragmentos de código implementam cada feature do FM atual, usando a arquitetura de referência como um artefacto intermediário.

A abordagem EvoSPL é suportada por alguns níveis de automação. Especificamente, a fase de mapeamento é suportada por uma ferramenta chamada friendlyMapper. A avaliação da abordagem EvoSPL é realizada no domínio automóvel, usando um estudo de caso de tamanho industrial na Bosch Company. Uma avaliação quantitativa e qualitativa foi usada para avaliar a abordagem num contexto industrial real. Além disso, a avaliação demonstra as capacidades da ferramenta friendlyMapper para realizar o mapeamento com sucesso. Os resultados revelam que a abordagem EvoSPL é útil para apoiar de forma eficaz e eficiente uma adoção e evolução de uma SPL num exemplo real da área automóvel.

Palavras-chave: modelos de recursos, re-engenharia, linhas de produtos de software.

EvoSPL: An evolutionary approach for adopting software product lines in the automotive industry

Abstract

Companies often develop and evolve a products family, which creates the opportunity to reuse and differentiate the products in the family. In practice, the development and evolution of such products are typically performed in an ad-hoc manner. Thus, a systematic reuse is required. Software Product Lines (SPLs) have largely been adopted by industry as a key idea for systematic reuse. To migrate the existing products of a family into an SPL, especially in the automotive industry, practitioners still lack an end-to-end approach.

In this thesis, we present an evolutionary approach, named EvoSPL, for managing consistently the evolution of SPLs in the automotive domain. The EvoSPL approach adopts a systematic re- engineering process that is composed of three main phases: reverse engineering, forward engineering, and mapping. The reverse engineering phase captures the commonality and variability of existing products in the (current) Feature Model (FM). In addition, this phase contributes with a feature identification method and feature model derivation method. The forward engineering phase bootstraps the remaining products of a family into the SPL, and then handles its evolution whenever a new individual product is requested. The mapping phase relates each feature of the current FM to its locations in the implementation code, using the reference architecture as an intermediate artifact, which helps to propagate the changes from the higher levels of abstraction (FM) to the lower ones (code), while preserving the consistency between them.

The thesis is an industrial research describing an approach that handles the evolution of SPLs in a setting where the domain architecture is common within a products family. However, we believe that the EvoSPL approach could be applicable and useful in other industrial domains that fulfil the conditions of our approach. The mapping phase is supported by a tool called friendlyMapper. The evaluation of the EvoSPL approach is performed in the automotive domain using an industrial- sized case study at Bosch Car Multimedia company. A quantitative and qualitative evaluation was used to evaluate the approach within a real industrial context. Moreover, the evaluation demonstrates the friendlyMapper tool capabilities to perform the feature mapping successfully. The results reveal that the EvoSPL approach is useful for software engineers to effectively and efficiently support an SPL adoption and evolution in the automotive domain.

Keywords: feature mapping, feature models, re-engineering, software product lines.

Abbreviations

AR	**A**tomic **R**equirement
AUTOSAR	**AUT**omotive **O**pen **S**ystem **AR**chitecture
CAN	**C**ontroller **A**rea **N**etwork
CSDT	**C**lassical **S**ensor **D**evelopment **T**eam
CSVF	**C**lassical **S**ensor **V**ariants **F**amily
DF	**D**egree of **F**reedom
EMF	**E**clipse **M**odeling **F**ramework
FL	**F**eatures **L**ist
FM	**F**eature **M**odel
FODA	**F**eature **O**riented **D**omain **A**nalysis
IDE	**I**ntegrated **D**evelopment **E**nvironment
LOC	**L**ine **O**f **C**ode
NFM	**N**aming **F**eatures **M**ethod
NL	**N**atural **L**anguage
NLP	**N**atural **L**anguage **P**rocessing
PLA	**P**roduct **L**ine **A**rchitecture
PVS	**P**rototype **V**erification **S**ystem
RS	**R**equirements **S**pecification
RSD	**R**equirements **S**pecification **D**ocument
SD	**S**tandard **D**eviation
SMD	**S**ingle **M**aster **D**ocument
SPL	**S**oftware **P**roduct **L**ine
SPLE	**S**oftware **P**roduct **L**ine **E**ngineering
SRD	**S**oftware **R**equirements **D**ocument
VD	**V**ariability **D**ocument
VK	**V**ariability **K**nowledge
VR	**V**ariability **R**ules

List of Publications

Conferences:

[1] Karam Ignaim, João M. Fernandes, André L. Ferreira, and Jana Seidel, " Systematic Reuse-Based Approach for Customized Cloned Variants," in proceedings of the 2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC), vol. 34, no.7, pp. 287–292, 2018.

[2] Karam ignaom, João M. Fernandes, "An Industrial Case Study for Adopting Software Product Lines in Automotive Industry: An Evolution-Based Approach for Software Product Lines (EVOA-SPL)," in proceedings of the 23rd International Systems and Software Product Line Conference-Volume B, pp. 183–190, 2019.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 General information

Some companies in the market need to handle multiple products that have some characteristics in common [Alv+10]. In fact, many products in a specific industrial domain have the same application architecture, the same execution platform, and a similar market segment [SCC16]. A products family typically specifies a set of similar products that are derived from a common platform and possess specific functionalities (features) to meet particular customer requirements. Each member of a products family is called a product. As a products family (i.e., all its products) can satisfy a certain market segment, each product is developed to address a specific set of customer needs within that market segment [LSR07].

The products of a family share a set of common assets, while each product contains a variable asset that makes it unique [LSR07]. A new product can be built by reusing the shared assets to obtain benefits like a shorter time-to-market and a higher quality, provided that these assets have already been developed, tested and used. An asset refers to any artifact that is part of the software development process, such as a requirements document, an architecture specification, a product description, a domain model, and programming code [AR12]. Thus, not only code can be reused, but other artifacts can also be reused.

Software Product Lines (SPLs) aim to support the development of a whole family of products through systematic reuse of the shared assets [CN01]. Companies consistently report that SPLs yield significant improvement in productivity, time to market, quality, and customer satisfaction [GM09]. Whenever a set of products is developed in a setting where ad-hoc reuse practices are common (i.e., without an explicit concern with respect

to variability and commonality), we consider that they do not constitute an SPL. The family of products in an SPL share a common set of features while allowing variability to satisfy different customer needs [Alv+10] [GM09] [Ra 3] . Variability allows choosing between different options, which we call variants. The variability is presented over all the artifacts of an SPL, including for example the requirement documents [Dav+13] [Ach+12] [Rab+10], the design models [EBB04] [Alf+08] [BM07], and the code [ES+12] [ZCA17].

SPL engineering (SPLE) provides concepts, mechanisms, and tools to support the developments of SPLs. In practice, the introduction of SPLE often begins when a given company finds itself with a successful family of products [LSR07]. Typically, traditional development approaches apply ad-hoc mechanisms to introduce the variability. For instance, a common and simple way to create products is to copy and adapt the existing products using a clone-and-own approach [RCC13]. Clone-and-own approaches promise low initial costs for creating products as the existing code is easily reused. However, they also come with major drawbacks for maintenance and evolution, since changes need to be synchronized among several products of a family [Dub+13] [RDR03]. SPLs provide solutions to these problems, because commonality is defined only once and the change in an SPL also needs to be applied only once. Therefore, the migration of a set of cloned products into an SPL would be enough solution to replace such ad hoc-reuse [Fen+17] [Kru01].

Commonality denotes features that are part of each product in exactly the same form. Variability enables the development of customized products, by reusing predefined assets. Hence, variability of an SPL distinguishes its products. Example 1.1 illustrates what we mean by commonality and variability.

Example 1.1: commonality and variability in the mobile phone system
The ‘mobile phone’ products family offers users a common ‘connectivity’ feature. This feature is part of each mobile phone product sold to any customer. At the same time, it offers users a choice of connectivity (options/variants). The customers of a ‘mobile phone’ product can choose the connectivity type that includes ‘Bluetooth’ or ‘WI-FI’ of the ‘mobile phone’ product when required.

Due to the need of dealing with large size and complex product families in the long-term, the companies must handle them in a systematic fashion [BP14]. A prerequisite for systematic reuse in SPLs is the ability to identify commonality and variability in terms of

features, which helps to specify explicitly the evolutionary changes for the products of a family [MP14] [MP07] [bastarrica2019software]. In this context, commonality and variability of a set of products are managed in terms of features. A feature represents an increment in the functionality that is useful and valuable to some stakeholders. Features describe the functional (the main target of our approach) as well as the quality characteristics of the systems under consideration [Fen+17]. A feature in an SPL is in a single place, but shared amongst many products, hence, changes to the implementation only must be performed once. Consequently, the effort for synchronizing product variants in an SPL is reduced.

Feature modeling is a well-established means for the domain engineering of a products [Ra 3] [Dav+13] [ZCA17] [Sch+12]. Normally, FMs are widely used to capture the common and variable features of SPLs. It leads to a clearer definition of the features supported by an SPL and the dependencies among them. FMs have been suggested as a suitable technique to abstractly describe the overall migration and evolution of the SPLs. Besides, an FM models the commonality and variability of a given SPL by specifying in a parent-child structural hierarchy features of its products [Dav+13]. FMs are typically used to generate and to validate the individual product configurations and to provide support for the domain analysis [Dav+13] [Ach+12] [ZCA17]. Depending on the abstraction level, a feature may refer to a prominent or distinctive user-visible characteristic or functionality of a product [Sch+12] [Dav+13]. Each valid configuration of features represents a product in the SPL, which includes a selection of optional features to accommodate the specific demands of customers in a particular domain [Ra 3].

1.2 Thesis domain

SPLs have been commercially applied in many industry domains [BP14], including embedded and automotive systems [Men+09] [RW05]. This research work deals with variability of a (software-based) products family in the automotive domain, which is implemented using the industrial C/C++ programming language. A survey reveals that C/C++ is the most popular and used programming language in the automotive industry [Men+09]. Normally, variability is realized in the code of such families using preprocessing directives, which cause a high complexity in the code [RW05]. In consequence, the products of a family get more difficult to understand and to maintain, which can easily lead to inconsistent changes and high complexity in the family.

The automotive industry faces many complex challenges in variability management, due to its hard products domain [RE14]. In addition, the approaches used in the automotive domain tend to extend existing functionality in an evolutionary manner rather than to develop a new functionality from scratch [WW02]. Thus, numerous research papers point out the necessity to use requirements to manage the variability. Unfortunately, most of them do not consider or give attention to requirements documents written in a NL [Men+09]. Even though such documents are more favorable to be used to manage the variability of a products family [LSS17], as they enable to manage and map variability from the beginning of an SPL life cycle. Hence, we present the EvoSPL approach to close this gap.

1.3 Motivation

There are three main factors that are important in SPLE [GM09]. The first factor is commonality and variability management in the same series of a large family of products. It refers to eliciting and communicating the commonality and variability in requirements to stakeholders. The second factor is traceability of commonality and variability from requirements to the code. It refers to relating between a requirement/feature (as relevant for variability management) and its implementation in the code. The final factor is managing and tracking reuse of the code across different products, usually driven by the previous two factors.

There is no single best approach that is suitable for all product families, since each family has a unique context that includes elements like scope, company, and market strategy [Wij03]. As these elements can change over time, the SPL approach must also change accordingly. To investigate the current state of SPLs evolution, a systematic literature review is presented in [SCC16]. The results show that there is no agreement about SPLs formalization; what assets can evolve, or how and when they evolve. Case studies are quite popular, but, unfortunately, few industrial-sized cases are publicly available. Also, few of the proposed techniques offer tool support.

Additionally, most approaches still lack support requirements written in a natural language (NL) [Alv+10]. Thus, there is a clear need to develop an approach that supports SPLs evolution using the requirements written in a NL. There is no clear consensus on how SPLs evolve, including which artifacts are considered and how they can evolve. It is

also not totally clear what formalisms, techniques, and methods should be used to support the process [Alv+10]. Thus, we conducted this research work to provide an approach that supports the SPLs evolution in a changing context. The approach is intended to be used to manage evolution for both the SPL and its products (existing products of a family) as well as the new products that arise throughout the SPL lifecycle.

1.4 Challenges in the research topic

Generally, companies apply an approach called clone-and-own mechanism (also referred as cloning or branching) to create a new product (see Fig. 1.1). With this approach, the artifacts of existing products are copied to be the basis of a new product, which are then modified. Cloning requires no major upfront investments and it is straightforward. This makes it a common and flexible software customization technique in the short term [SCC16].

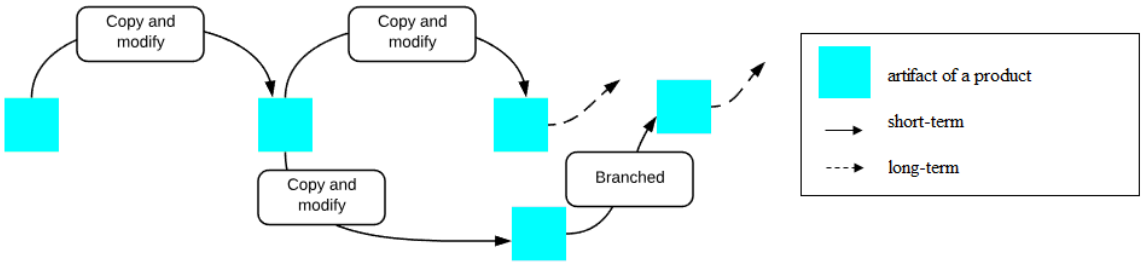


Figure 1.1: Cloning and branching approaches.

However, cloning does not favor reuse in the long term, and it has its downsides. When the number of products increases in a family, changes can have a complex impact on the entire family and inconsistencies are unavoidable and may lead to further inconsistencies until the changes are propagated to the entire products of a family. All this makes the maintenance of the products or the family much harder [Ra 3]. This increases complexity and cost when developing various products and puts the developers into problems, such as the inconsistency of a family and finding the implementation of a feature in the code [Bor09].

SPLs have proven to help the companies in organizing product families at a low cost and in the short term with high quality [RCC13] [Tan+10]. In practice, the developing organizations face a choice between starting SPLs from scratch or considering approaches

which evolve SPLs when migrating the existing products (after their success in a market segment) [Fis+14] [Dus15] [Mac+14] [Kru01] [Alv+05].

The adoption of an SPL-based approach in the automotive industry has many challenges. Initially, migration of the multiple products into an SPL often requires commonality and variability to be considered [MP14], because they are important concerns in a family development [JT00]. Hence, we identify challenge 1:

"it is required to identify commonality and variability of the product family that will form an SPL".

Once this has been completed, challenge 2 is important

"the variability between the products needs to be captured using specific variability models" [MP07].

This model defines the available variants. The features represent an SPL in a way that is meaningful to different stakeholders, which make FMs a suitable candidate to describe an SPL and support its overall evolution [MSC14]. Besides, FMs are popular SPL assets that describe the commonality and variability of a family of products [Dav+13]. Thus, the industry has successfully adopted FMs for very large SPLs [BCH15]. For example, all mobile phones allow the users to make calls. Thus, this feature is obviously shared among all the mobile products and constitutes a commonality for them. The mobile phones allow different connectivity methods like 'Bluetooth' and 'WI-FI' and this differentiates the product in some way with respect to the others in the same family [LSR07] [Ra 3].

In the automotive domain and probably in many other domains as well, the considerations that influence FMs derivation is extremely complex and, at the same time, need to be documented as clearly as possible for later reference [RW05]. Furthermore, when it comes to complex systems, specifically in the automotive domain, managing such systems just using the requirements documents becomes nearly impossible [WW02]. Many research address management of the variability at the design-code-levels [Ra 3] [BM07] [She+11] [vauttierdocumenting] [urtadodocumenting] [Zia+12] [AT02] [Wan+09] [CD12] [JB08]. Thus, we identify challenge 3:

"managing variability, especially at the requirements-level is still a core issue and has received less attention" [IRB14] [LSS17].

Requirements engineers in the automotive domain aim to specify requirements at a detailed and technical level [Bra+14]. At the same time, variability is available in the requirements and system specification [Men+09]. So, relevant information can be extracted from these requirements [Ape+18]. After a long-term investment, software producers need to migrate more products to an SPL. They need to regularly update the SPL whenever new specific market requirements arise. Therefore, we present challenge 4:

"software engineers need to evolve an SPL to reflect new and changed requirements for its products or to add a new product (not currently covered by the SPL)" [Bor09].

This can require changes in the whole SPL. For that, software engineers need to handle the evolution of an SPL in a systematic way. During the evolution of a given SPL, it is necessary to refactor some artifacts, such as FMs and code [ZB12]. For instance, in the automotive industry, managers constantly make decisions about future evolution, like "the ABS warning lamp is introduced with the new product". With a rising number of features, feature changes, and evolution steps, the systematic evolution of an SPL becomes essential. Earlier works suggest FMs as a suitable means of abstraction to describe the overall evolution of an SPL. So, our challenge 5 is:

"there is only very little support for evolution of the feature-oriented modelling of an SPL so far" [Bot+10].

Evolving SPLs is risky because it might impact many products [Nev+11]. So, during evolution to transform the products into an SPL or introduce new products, which gives rise to challenge 6:

"during evolution of an SPL, it is important to make sure that the behavior of existing products is not affected".

Thus, further processing like refactoring is necessary. It is a good practice to extend the definition of refactoring (usually applied to software programs) to be applied in context of SPLs [Ach+11] [Alv+06b]. The traditional notion of refactoring does not handle appropriately the variability. Such traditional refactoring notion does not handle FMs appropriately, or transformations involving multiple products of the same SPL. Also, it does not guarantee the configurability improvement in an SPL. For that, we consider challenge 7:

"an appropriate variability-aware refactoring technique could be used to transform the multiple products into an SPL or to refine the SPL after it has been established at the feature model-level" [Alv+06b] [Lie+15].

Refactorings or refinements specify several change operations to the structure of an SPL, to improve (maintain or increase) its configurability without changing the observable behavior of its original products. The changes include not only the FM itself, but also the mappings and associated assets. For instance, a new mandatory feature can be safely added to an FM, if it is already part of all products. The same feature can be converted to an optional feature later.

In the context of an SPL evolution, many change requests are raised over time and all of them must be integrated into an existing SPL in a consistent and coordinated way. Thus, challenge 8 is:

"it is required to define links between the related artifacts of an SPL".

For instance, we need to map features to their implementation [ES+12] [XXJ12] [ESSD14]. This problem is addressed by the feature location, which leads to challenge 9:

"software engineers need to identify and trace variability locations across the scattered code of a products family".

To work towards this, it is not only important to understand the code that is relevant to an SPL and locating the variability (features) in this code, but also to consider challenge 10:

"to refine the FM and mapping links when the changes occur".

In accordance with the challenges described above, this thesis proposes an approach for SPLs evolution. This approach includes activities taking care of migration and change of a products family. We propose the EvoSPL approach concepts based on the results of the review introduced by Marques et al. in [Mar+19] and based on investigation and discussion of a sample products family in the automotive system at Bosch Car Multimedia (the Bosch company at Braga).

1.5 Objective

The main objective of this Ph.D. thesis is to propose an approach (EvoSPL) that helps to adopt SPLs in the automotive industry. The approach aims to help companies in the automotive industry (1) to evolve a given SPL, focusing on migration of the existing products of a family that were delivered to customers in the past, and (2) to use the bootstrapped SPL as a base to evolve it with new products, to get the resulting SPL. The approach describes a re-engineering process towards managing the evolution of SPLs in a setting where the domain architecture (i.e., reference architecture) is common to a family of products, which was developed without considering a systematic approach. The approach explores how variability of a products family can be managed at the requirements-level and how the mapping of features to different types of artifacts (requirements document, reference architecture, and source code) can be established. The approach provides a systematic process with some level of automation.

The EvoSPL approach offers the software engineers a stepwise process to switch from scenarios where the products of a family are managed with ad-hoc approaches into scenarios where commonality and variability are explicitly addressed using variability models. There are several variability models that could be applied here, and in this research work, we use an FM.

This Ph.D. thesis is an industrial research, describing a case where we handle the variability of a products family, developed to satisfy a range of customers in the automotive industry. The use of an industrial-sized case study allows us to validate the work in what concerns the usefulness, effectiveness, efficiency and applicability of the proposed approach. The results provide initial positive insights about the approach with respect to those criteria.

1.6 Problem statement

SPLs evolve over time. One practical problem is that during SPLs evolution, (1) the variability and interdependencies between products should be taken into consideration. In addition, (2) the artifacts, especially the code, become difficult to understand, to use, and to maintain. Typically, more problems arise over time with an implicit or already lost about the consistency and the supported variability.

For decades, C has been the preferred language for developing automotive industrial systems. The variability mechanisms offered by the C preprocessor directives are simple. The experience with large-scale and evolved SPLs shows that the respective variability tends to become overly complex, due to the variability realization with conditional compilation (#ifdef blocks). In cases of extracting the variability from such automotive families, the C preprocessor directives can be easily abused, leading to situations in which the code is hard to understand, and all possible individual products are extremely difficult to analyze [TR13]. A support for the variability at code-level is provided by a small number of concepts and tools [Men+09], but they do not consider the specific requirements of the automotive domain.

With respect to the challenges to support the initiation of an SPL from the set of existing products family and to support its evolution (the main objective, which exists due to the lack of evolutionary approaches), this thesis is motivated by the following problems.

P1. To analyze and identify the similarities and differences among multiple products of a family at requirements-level.

P2. To model the commonality and variability of a products family and its evolutionary changes (at a higher level of abstraction) in a systematic way, using a variability model.

P3. To bootstrap the existing products of a family into an SPL.

P4. To evolve the already established SPL to encompass new products, keeping the existing products valid in the SPL.

P5. To use a proper refactoring technique in an SPL, so that it considers new requirements but still considers the previous ones.

P6. To preserve consistency among artifacts involved in the SPL evolution. This requires one to establish a traceability that maintains the consistency among requirements, features, and code. Thus, features need to be mapped to their places in the artifacts of an SPL (especially the code), allowing the changes of an SPL at higher levels of abstraction (e.g., an FM) to be propagated to lower ones (the code).

As these problems accumulate, it is worth handling evolution in a systematic way, by explicitly specifying the evolutionary changes of a set of existing products family into

the SPL and by integrating new requirements that arise during the new product request directly into the SPL [Ach+12].

1.7 Hypotheses

In accordance with the problems stated above, our thesis formulates the following main hypothesis.

Main hypothesis. It is possible to recommend an approach that supports a re-engineering process to migrate a family of products (using its requirements document written in NL) into an SPL. In addition, it is possible to evolve the SPL after it has been established. This hypothesis addresses the main objective of this thesis. To confirm this main hypothesis, it is required to prove the hypotheses from H1-H6. Fig. 1.2 presents the relation between the research problems and the hypothesis.

H1. It is possible to develop a semi-automatic difference analysis approach that identifies the differences between the requirements documents of two products that are parts of the same family.

H2. It is possible to specify commonality and variability among the members of a products family (e.g., as feature modeling) in the automotive domain with explicit variability model (i.e., an FM).

H3. It is possible to capture and specify the observed changes in the requirements of existing products of a family in a systematic way, during analysis of evolution of an SPL.

H4. It is possible to specify changes in the requirements of a new product during an SPL evolution (i.e., to extend an SPL with the new product, whenever there is enough implementation similarity among the products and a new one).

H5. It is possible to apply the appropriate refactoring technique, to bootstrap an initial SPL from the existing products, and to extend the bootstrapped SPL to encompass another product, at the feature-model level.

H6. It is possible to relate the artifacts that are relevant for variability management, like mapping requirements documents and an FM to its documentation in the reference architecture, and to its implementation in the code, while preventing inconsistencies among them.

1.8 Contributions

The thesis presented in this research work contributes with an approach named Evolutionary Software Product Line (EvoSPL). The approach supports SPLs evolution using migration of the existing products into an SPL and considering the changed requirements for its products. The contribution of this thesis is rooted in the issues that were identified in the products family from the automotive industry that the author investigated at Bosch Car Multimedia company. Additionally, based on the concepts, foundations, and results of the review that appears in [Mar+19].

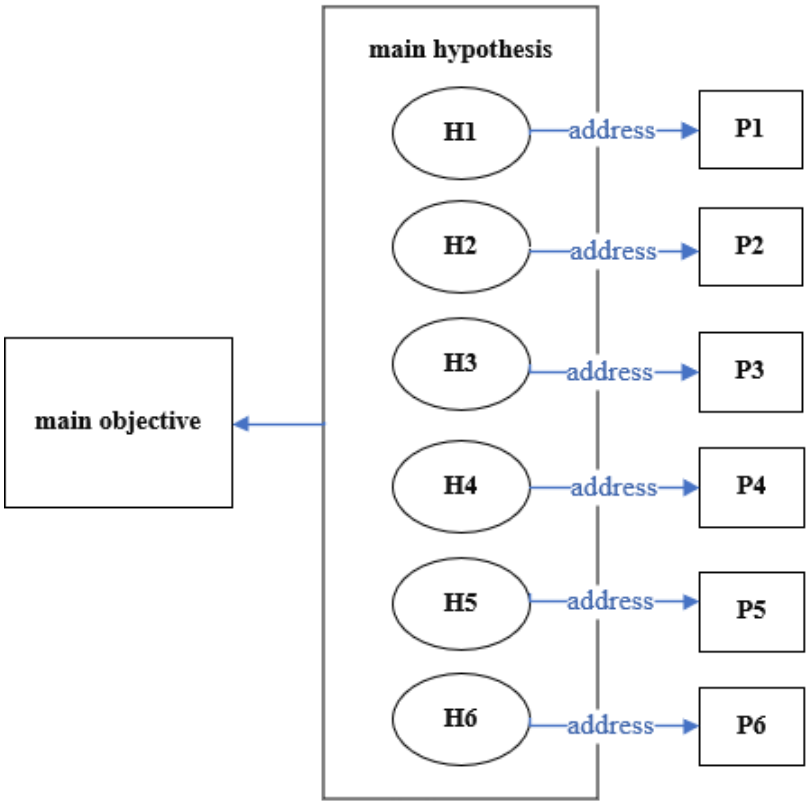


Figure 1.2: Thesis hypotheses mapped to the research problems.

This thesis contributes with an approach to SPLs evolution that uses a variability modelling, a catalog of change (evolutionary) operators, and a semi-automatic tool to support

the mapping that guarantees the consistency between an FM and the code of a products family. The contributions of this thesis are incorporated in the approach phases, as well as in the evaluation case study. This helps to observe if it is possible to use the EvoSPL approach in solving the problems that are introduced in section 1.6.

Our research contributes to the knowledge of software engineering related to SPLs evolution using systematic reuse by providing the following characteristics.

- A difference analysis designed to fit product families is provided that analyzes and compares the multiple products of a family (two different versions at the same time). It accepts as input requirements documents written in NL (were delivered as a side result of ad-hoc reuse) and identifies the similar and variable requirements between the two documents at once. As previous works have shown great potential in the requirement artifact of SPLs, thus, choosing them for a difference analysis approach is viable. Furthermore, the difference analysis results in the change terms of requirements, for performing variability analysis of a products family.

- A variability analysis dedicated to the EvoSPL process is presented that identifies commonality and variability (among the products of a family) in terms of features. It analyzes the variability of requirements documents to identify common and variable requirements, to support their presentation in terms of common and optional features. Furthermore, it identifies relationships among the requirements in terms of dependencies among features. The variability analysis introduces novel feature identification method and an FM construction method.

  Both contributions (mentioned above) allow to capture the variability to a variability model (i.e., an FM), which influences the migration and the evolution processes.

- A stepwise SPL bootstrapping, which guides the software engineers to bootstrap the remaining products of a family into an SPL, is presented. Besides, an SPL evolution, which guides the software engineers to evolve the family with a new product (upon new customer request), is presented as well.

- A refactoring-based scenario that supports a concept for specifying refactoring of FMs is introduced. Part of the scenario adopts a work that presents a set of sound refactorings for FMs in SPLs context [Alv+06b].

- A novel feature mapping solution of an SPL is developed. This solution is based on the use of the reference architecture as a centric point, for tracing variability

from features to the code. For instance, the mapping between an FM and its implementation code, as well as maintaining the mapping of an FM and code whenever features change occurs. Besides, mapping features to the lines code, especially if they were only derived from requirements, is a non-trivial task.

Thus, the feature mapping presented in this thesis: (1) defines and maintains the traceability definition in a backward direction from a feature to the requirements that specify this feature, by taking into account the specification of each requirement and (2) traces features to the different types of artifacts (especially to the implementation code) of a products family. In combination, those traces specify the complete traceability definition between features to the requirements specification, the reference architecture, and the code.

- A tool named friendlyMapper supports the mapping phase of the EvoSPL approach presented in this thesis. The tool aims to support the changes. It automates the mapping between features belonging to the FM to the code fragments belonging to the code. The tool uses traceability links of the traceability tree. These traces are established between each feature of the FM and a set of code fragments that satisfy this feature. This tool helps to propagate the changes at higher-levels of abstraction (requirements and FM) to lower one (code) and to preserve the consistency between them.
- The contributions of the EvoSPL approach have been evaluated in an industrial-sized case study with a products family of the automotive industry. The evaluation contributes to apply and validate with a real set of products family. Within the industrial case study, which has been conducted at Bosch Car Multimedia company, a qualitative analysis, including interviews, observations, and surveys have been applied. Furthermore, a quantitative analysis using an empirical treatment has been applied. The evaluation investigates the applicability of our approach for a real set of customer products and gets feedback from the software engineer at the company.

1.9 Thesis structure

The remainder of this thesis is structured as follows.

- Chapter 2 introduces the background and foundations of this thesis. It discusses SPL basic concepts and the related refactoring technique.

- Chapter 3 presents the approaches related to the topic of this thesis for better understanding the state-of-the-art in this area.
- Chapter 4 provides an overview of the EvoSPL approach. It presents the structure, input, output, and key concept of the approach.
- Chapter 5 presents the reverse engineering phase, including the appropriate difference analysis and variability analysis activities that are dedicated for SPLs. Moreover, the chapter introduces the feature model synthesis activity that represents the variability among a set of products belonging to the same family in a systematic way, which leads to facilitate the migration process.
- Chapter 6 presents the forward engineering phase. This chapter explains bootstrapping and evolution of an SPL. The former involves adding a set of available products into an SPL, and the latter extends the SPL to encompass a new product. For that, this chapter educates refactoring concepts that are aware of an SPL context.
- Chapter 7 presents the mapping phase; it describes a technique to trace features to their places in the code. Moreover, it explains how to deliver a reference document to store the mapping results.
- Chapter 8 describes the evaluation of the EvoSPL approach, using an industrial-sized case study, including different evaluation types to prove correctness of the thesis-hypotheses.
- Chapter 9 concludes the thesis by providing a summary for the EvoSPL approach and suggestions for future improvement.

Chapter 2

Background

This chapter presents the background needed to understand the proposed approach. It uses the concepts, practices, approaches, and contributions from the valuable topics of mainly software engineering and informatics. The following sections introduce these topics categorized in re-engineering, software product lines, and natural language processing. In addition, this chapter presents techniques and concepts related to our approach and used during the evaluation.

2.1 Re-engineering

Re-engineering is an important activity in software engineering to prevent software systems from turning into legacy systems and losing value over time [DDN02]. When we need to update the software to keep it up to date, without impacting its functionality, it is called software re-engineering. It is a thorough process where the design of software is changed, and programs are re-written. Re-engineering is "the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form", as defined by Chikofsky and Cross [CC90].

As shown in Fig. 2.1, re-engineering is a combination of reverse engineering and forward engineering [CC90]. Reverse engineering involves analysing a given system in order to determine its components and the relations between those components [CC90]. It also involves the creation of alternative representations of the system, usually at a higher level of abstraction. It can occur at any stage in the software development life cycle. Forward engineering is the process of moving through the stages of design, starting at the highest level of abstraction moving to a specific implementation [CC90].

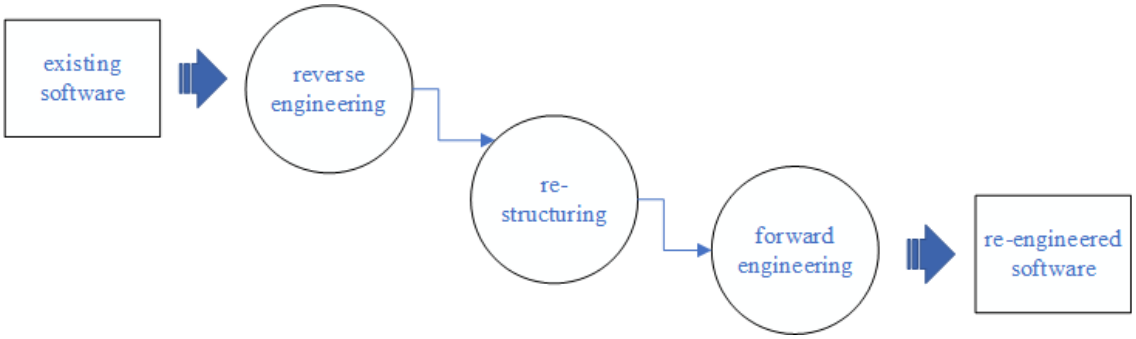


Figure 2.1: Re-engineering of software systems.

The concept of re-engineering is well-suited in the context of the SPLs. Rarely SPLs start from scratch. Instead, they usually start from a set of existing products that undergo a re-engineering process, as presented in [Alv+05]. Initially by using reverse engineering to extract variation from existing products and then by using forward engineering to adapt the newly created SPL (called in this thesis the resulting SPL) to encompass other variant products. Existing approaches extract FMs from a high level of abstraction (e.g., product description, requirements, etc.) [IRB14] [Dav+13] and from a low level of abstraction (i.e., source code) [AM14] [Zia+12].

2.2 Software product line and products family

A products family refers to a set of similar products that are derived from a common platform and possess specific features (functionality) to meet particular customer requirements. Each individual member within a products family is called a product [KS98]. SPLs are used in industry to develop a family of similar software products from a common set of shared assets. An SPL is "a set of software intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets (artifacts) in a prescribed way" [CN01].

An SPL is now a popular approach for variability management and software reuse in software engineering [Mar+17]. Software reuse is the process of creating new products from existing ones, rather than building products from scratch [Kru92]. SPLs have received increasing attention within the software engineering community, especially from industry. An SPL can be seen as a family of products that have been developed with an explicit concern about commonality and variability, during the development process [Ra 3].

SPLs have been used successfully by industry to promote reuse. For example, several reports from large companies such as, Bosch, Nokia, and Philips observe benefits with their use, especially with respect to the reduction in time to market [Wan+12]. Because of long term dealing with product families, companies need to handle them in a systematic way. Therefore, SPLs can be a suitable option in this case.

Software product line engineering SPLE provides concepts, mechanisms, and tools to support the developments of SPLs. It has proven to empower organizations to develop a diversity of similar products at lower cost, in shorter time, and with higher quality, when compared with the development of single systems [PBDL05]. SPLE is an inter-disciplinary concept to develop software applications (software-intensive systems and software products) using platforms and mass customisation. A platform is any base of technologies on which other technologies or processes are built. Mass customisation is the large-scale production of products tailored to individual customers' needs [PBDL05].

SPLE Process SPLE is a two-phases approach which consists of domain engineering and application engineering, as shown in Fig. 2.2, the domain Engineering handles the common parts among the similar products of a family. The main task of domain engineering is to build the SPL architecture consisting of a core asset and the software variant features, while the application engineering focus on the derivation of the new products by the different customizations of software variant features applied onto the core asset [Ape+18] [Ape+13]. While commonalities and variabilities are handled mostly in domain engineering, product-specific parts are handled exclusively in application engineering. This is shown in Fig. 2.2.

Evolutionary SPL adoption There are two different approaches for adopting an SPL approach: revolutionary and evolutionary SPL adoption, according to Bosch [Bos02]. In the revolutionary approach, a company introduces an SPL from scratch that is designed as a set of SPL product members and further members that are planned for the future. In the evolutionary approach, the SPL is derived from a set of existing product members (e.g., the EvoSPL approach proposed in this thesis). The revolutionary approach allows taking future product variants into account and planning for a broader scope of the SPL. In contrast, the evolutionary approach allows for a more focused and less complex adoption of the SPL approach.

Variability management In an SPL, variability management is a key activity that usually affects the degree to which it is successful. SPLE exploits variability management (the

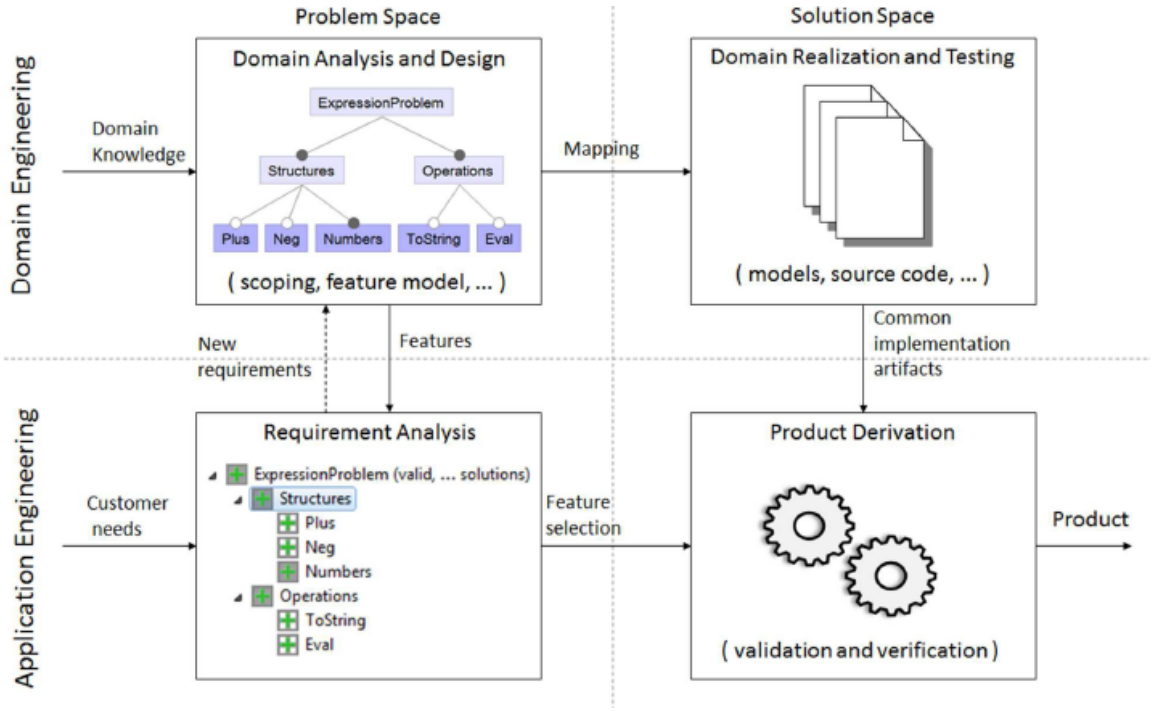


Figure 2.2: Overview of an engineering process for software product lines [Ape+13].

commonality and variability) of the products that belong to an SPL and systematically handles the variation (i.e., the differences) among those products [MP14]. This is achieved through the identification and management of commonality and variability in the artifacts of a set of products, such as requirements, architectures, components, and test cases. Variability management defines and manages the commonality and variability among the members of a products family. At the same time, it represents a major challenge during the development and evolution of SPLs [SSW09].

Commonality is a property shared by all the products of the SPL [MP14]. For example, all mobile phones allow users to make calls. Variability of the SPL defines how the different products of the SPL can vary [Met+07]. Products of the SPL may differ in terms of features and requirements they fulfil. For example, some mobile phones may include mobile Bluetooth connectivity, others not. Also, the 'e-shops' SPL that supports 'search', 'order product' and 'return product' features. The 'search' and 'order product' are common features that their specification (in requirements document) and implementation (in the code) are required to exist in all the products of the 'e-Shop' SPL. The 'return product' is an Optional feature that may appear or not in the specification and implementation of the products [IRB14].

Feature models FMs are one of the most popular models for describing variability of an SPL [Dav+13]. An FM documents the features of an SPL and their relationships. Features constitute the basic building blocks of FMs. Depending on the level of abstraction and artifacts described, features may refer to a prominent or distinctive user-visible characteristic of a product (i.e., an increment in a software code base) [Ape+18] [Dav+13]. In the context of this thesis, a feature represents an increment in functionality that is important to some stakeholder.

A recent survey of variability modelling revealed that FMs are by far the most frequently used notation in industry [Ber+13]. Academics, researchers, and industry have developed tools to specify them graphically or textually and automate their analysis, configuration or transformation [Beu12] [Pur] [Big]. FMs were first introduced in the Feature-Oriented Domain Analysis (FODA) method by Kang in 1990 [Kan+90]. Feature diagrams define an FM as a hierarchy of features and constraints among them. A feature diagram is a graphical notation to specify an FM. It is a tree whose nodes are labelled with feature names [Kan+90].

To switch to SPLE starting from a collection of existing products, the first step is to extract the FM that describes the SPL. This implies to identify the products family's common and variable features. Reverse engineering of an FM for a products family is essential [Zia+12]. Variability should be expressed through a relationship between two features. The features of an FM are [Wan+12]: (1) relationships of variability in the given domain (the Optional feature 'Camera' of Fig. 2.3) and (2) relationship of dependency (the 'Camera' feature requires the 'Bluetooth' feature of Fig. 2.3). The semantics of variability and dependency relationships on an FM are described in Fig. 2.4). It is worth to mention that we called relationships of variability and relationship of dependency in this thesis as parent-child relationship (including variability-pattern of the feature) and dependency-relationship respectively. An FM defines which feature combinations lead to valid products within the SPL. The individual features are depicted as labelled boxes and are arranged in a tree-like structure.

Fig. 2.3 shows a small example of the FM of an SPL from the 'mobile phone' domain (related to Example 1.1). The root feature of the tree is called 'Phone'. There is always exactly one root feature that is included in every valid program configuration (or tree). A feature is said to be Common (or mandatory) if it is presented in all the products of a family. Contrarily, a feature that does not appear in all the products of a family is called

Optional [BM07]. The FM supports the common 'Connectivity' feature with optional support for 'Camera'. The 'Connectivity' feature is available either as 'Bluetooth' or as 'WI-FI'. Features can also be grouped in 'Or' / 'or-groups' (like 'Bluetooth' and 'WI-FI') or 'Alternative' / 'xor-groups' (i.e., alternatives relation). The dependency relationships can be defined between features in the model like (1) 'requires' (selecting a feature requires to select another feature), for example, 'Camera' requires 'Bluetooth', or (2) 'excludes' (two features mutually exclude each other). [Dav+13] [UV] [She+11].

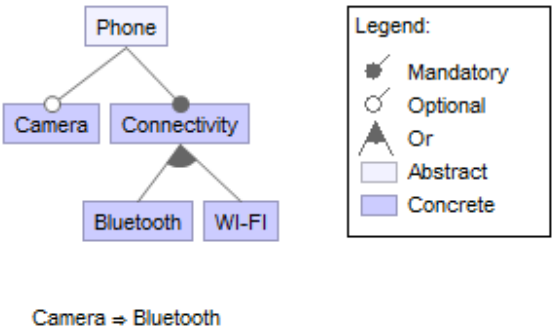


Figure 2.3: The feature model example from the mobile phone domain.

relationship	type	notation	semantic
variability relationship	Common\ Mandatory		If the father feature is selected, the child feature must be selected as well
	Optional		If the father feature is selected, the child feature can but need not to be selected
	Or		If the father feature is selected, at least one feature of the or-child-features must be selected
	Alternatives		If the father feature is selected, exactly one feature of the alternative-child features must be selected
dependency relationship	Requires	⇒	If one feature is selected the implied feature has to be selected as well, ignoring their position in the feature tree
	Excludes	⇒ ¬	Indicates that both features cannot be selected in one product configuration and are therefore mutually exclusive

Figure 2.4: Relationship semantics of the feature model adapted from [ML04].

Feature location Feature location concept often well- known as feature mining or feature identification [AM14]. In general , feature location is the activity of identifying code units that implements functionality (feature) in a software system [Dit+13] , and it is one of the most important and common activities performed by developers during software maintenance [XXJ12] [ESSD14]. Feature location in a collection of a products

family consists in identifying a group of code units that constitutes its implementation. This group of code units must either be present in all products (case of a common feature) or in some but not all products (case of an optional feature) [Zia+12].

Feature location techniques mainly use different types of analysis, such as textual, static, and dynamic analysis [Ra 3] [Zia+12] [XXJ12] [ESSD14]. For achieving better results, hybrid approaches combine two or more types of analysis, like textual, static and dynamic analysis, with the aim of using one type of analysis to compensate for the limitations of another [Zha+06]. One more concept related to the feature location is traceability link presented below.

Traceability It defines and maintains relationships between artifacts involved in the software life cycle [GF94] in forward and backward directions. For example, it could be defined from requirements to code and from code to requirements. An important step in re-engineering the individual products of a family into an SPL, for systematic reuse, is to identify code units that implement a particular feature across products (see Fig. 2.5). This mapping between features and corresponding code units (called in our research as code fragments) is known as traceability links. Traceability links can be used to (1) bridge the gap between problem space and solution space of the SPL (see Fig. 2.6) and to (2) facilitate products derivation from SPL core assets [Val+17] [BBM05].

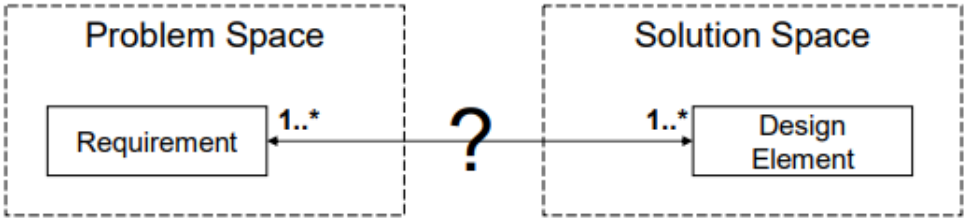


Figure 2.5: The problem and solution space of software product line engineer [BBM05].

In software systems with highly changed nature of requirements, traceability has become a critical issue. Thus, several research papers are presented over past years on traceability from problem space to solution space in traditional software development and evolution [DPG15]. This traceability is even more challenging in SPLE [PBDL05].

The terms problem space and solution space have been previously introduced in [BBM05]. The problem space is related to requirements. It specifies needs of the SPL domain and describes the features provided by the SPL from a customer perspective. The solution

space relates to implementation of the SPL and describes the variability in the program from the perspective of the developers [BBM05]. As shown in Fig. 2.6, the problem space generally refers to specifications of the products established during the domain analysis and requirements engineering phases, whereas the solution space refers to the concrete products created during the architecture, design and implementation phases. All the developed artifacts collectively form the SPL infrastructure.

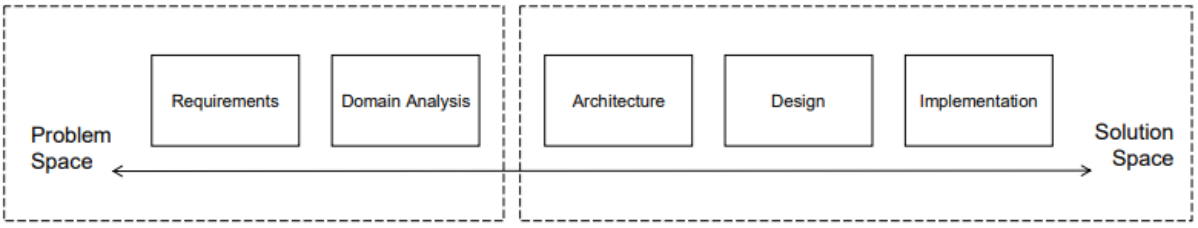


Figure 2.6: Traceability between the problem and solution space of software product line engineer [BBM05].

Refactoring Traditional refactoring (code refactoring or program refactoring) describes the task of restructuring the source code of a program to improve its internal quality without changing its external behaviour [CC90]. It can be considered as one task to perform in context of re-engineering [CC90]. Refactoring is usually motivated by noticing a code smell [Tuf+15], like an identical or very similar code (duplicated code) exists in more than one location. However, the traditional definition of program refactoring does not consider SPLs specific characteristics (e.g., SPLs configurations and FMs). Moreover, traditional refactoring usually transforms one program to another and does not consider merging the two programs. Thus, there is a need to consider refactoring in the SPL context, in which FMs are refactored, in addition to regular program refactoring [Alv+06b].

SPLs refactoring In context of the SPL, refactoring is a process which increases the quality of the reference architecture, an FM, and core assets of the SPL, but keeps its functionalities unchanged [THMH16a]. According to [Alv+06b], an SPL refactoring is "a change made to the structure of an SPL in order to improve (maintain or increase) its configurability, make it easier to understand, and cheaper to modify without changing the observable behaviour of its original products". An SPL refactoring involves not only program refactoring, but also an FM refactoring. An FM refactoring is "a transformation that improves the quality of an FM by improving (maintaining or increasing) its configurability" [Alv+06b].

SPLs evolution Because of long term, size, and complexity, companies dealing with product families and SPLs need to deal with evolution in a systematic fashion [SB99]. SPLs evolve when there are changes in the requirements (e.g., customers' requirements evolve continuously), product structure or the technology being used [bastarrica2019software].

Compared to single software evolution, SPLs evolution remains more difficult compared to single software evolution, the SPL evolution is more complicated because the SPL changes impact two levels: the level of products and the level of the core assets. Indeed, the requirements changes must be propagated in the two levels, which requires a well understanding of the change and accurate determination of its impact. Different approaches have been proposed for managing SPL assets and some also address how evolution affects these assets [SB99]. Further detailed explanation of SPLs evolution is presented in section 3.1, where we couple the topic with the related works.

Difference analysis It specifies the differences between different versions (variants) of an artifacts, such as approaches for program differencing (e.g., [AOH04]), common source code versioning systems, or text file comparison (e.g., [Difa] [Difb]) that compare text differences between two (or more) text-based files. A difference between two models can be used to compare two veirsions (variants) of models in terms of added, removed, and modified elements and stores the result in a difference model [AOH04]. Specifying changes is relevant in context of evolution and in context of variability, like to specify the differences among multiple products of a family in an SPL [IF19] [SHA12].

Change operators A change operator describes an operation performed on a model (e.g., an FM) to fulfil a change. Change operators are one of the basic concepts to specify changes. For example, Alves et al. [Alv+06b] and Seidle et al. [SHA12] define a set of change operators on FMs in context of SPL refactoring and evolution. The difference analysis concept can be combined with change operators to model evolution of an SPL, as presented in [Ple+12].

Migration towards SPLE In practice, a company starts thinking to introduce SPLE, when it finds itself with a family of products after some success in a market segment. In this case, when thinking of adopting SPLE, besides starting from scratch, the company can consider an approach that focuses on migration of the existing products of a family towards an SPL [Wag14]. Such approach is related to re-engineering (section 2.1), where assets (or artifacts) are extracted from existing products and re-engineered to contributes

a resulting SPL [Ass+17] [Ass15]. Those types of approaches that take existing products into account are mostly classified as an evolutionary approach [Bos00].

2.3 Natural language processing

Li et al, [DL18] define natural language processing (NLP) as a way for computers to analyse, understand, and derive meaning from human language. Using NLP, developers can organize and structure knowledge to perform tasks such as automatic summarization, translation, or relationship extraction. The objective of NPL is to let computers perform useful tasks involving human language, like enabling human-machine communication, improving human-human communication, or simply doing useful processing of text or speech [DL18] [AH06] .

NLP analysis NL analysis is a subtopic in the field of computer linguistics [ID10]. NLP analysis is the application of NL analysis to further extend the analysis of program structure and semantics. The analysis of a NL text may have three levels, as shown below. During these levels many techniques and tools can be applied by specialists.

- Level 1. Morphological analysis focuses on the words' structure [Tel00].
- Level 2. Syntactic analysis deals with the relationships between words in a sentence, deciding which classification group the word belongs to, according to a grammar, such as stems, root words, prefixes, and suffixes. [ID10].
- Level 3. Semantic analysis (or parsing) is built upon the results of the other two levels. It aims at defining words and sentences meaning based on the knowledge of their structure, relationships and role (e.g., studying meaning of individual word and the combination of individual words) [ID10].

Language processing To improve text analysis, there is a need for processing terms extracted from text [ID10]. On the following , we will shortly discuss the processing methods that have been proven to be valuable in the context of NLP in general [SKL08].

- Splitting involves separating strings into individual terms using information, like underscores, hyphens , capitals (CamelCase), or numerical digits ([0-9]). For example, splitting "getProductCopies" string to {"get", "Product", "Copies"}.
- Filtering involves removing useless words. For example, filtering {"get", "Product", "Copies"} to {"Product", "Copies"}).

- Stemming involves transforming a term to the stem of a word. For example, transforming {“Product”, “Copies”} to {“Product”, “Copy”}.

NLP in reverse engineering Li et all. [LSS17] briefly introduced how NLP techniques are generally used, with a specific focus on feature identification and variability extraction in SPLs. The general process presented below is shown in Fig. 2.7.

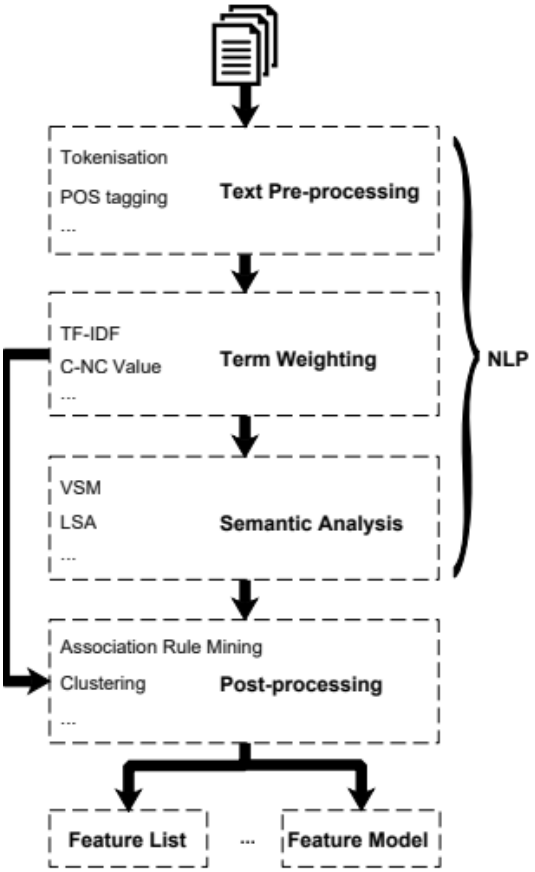


Figure 2.7: General process for applying natural language processing techniques in reverse engineering variability from natural language documents [LSS17].

1. Text pre-processing involves transforming NL documents into types of words that can be identified and analysed easily by computers. In particular, NL documents are divided into words, phrases, symbols, or other meaningful elements (e.g., using tokenization). Additionally, these elements can be tagged with their type of word, like noun, verb, and object. In addition, as stop words (e.g., ”the”, ”at”) lack any linguistic information, they are removed in this phase.

2. Term weighting which can be adopted optionally to estimate the significance of terms in NL documents, by calculating the frequency of their occurrence in different

NL documents. This can be performed using two commonly used techniques: Term Frequency-Inverse Document Frequency and C-NC value.

3. Semantic Analysis is an optional step that is typically used to gain semantic information. Several techniques can be used in this step. Certainly, Vector Space Model and Latent Semantic Analysis techniques are widely used to conduct a semantic analysis.

4. Post-processing performs further analysing of the transformed data in order to identify features and extract variability information. In this step, various methods can be used, such as, Clustering Approaches and Association Rule Mining. Cluster approaches are adopted to group similar features with a feature being a cluster of tight-related requirements. Association Rule Mining is used to discover agreements among features across products, and to augment and enrich the initial product profile. By the end of the post-processing, various outputs can be obtained, such as a feature list or an FM.

Text-based parsing In NLP, the initial step is parsing the sentence in a particular language (e.g., English) based on the grammar, in order to help understanding the meaning of a sentence and extracting a specific words (e.g., extracting the behaviours from the textual descriptions [IRB14]) [Sib+13]. Text-based parsing is a process that takes a given series of text and separates it into smaller terms or phrases conforming to the rules of a formal grammar. The application of text-based ranges from document parsing to deep learning NLP. The most important text parsing technique is called word tokenization.

Word tokenization Tokenization is the process of converting a text into smaller pieces of tokens based on certain rules (e.g., semantic roles [IRB14]). Using regular expression, own rules can also be created. Tokenization can be accomplished using an NLP parser and has an important role in text pre-processing tasks. For example, it helps in mapping parts of speech, finding and matching common words, cleaning text, and getting the data ready for advanced text analytics techniques.

Parser The program that develops the grammatical structure of the sentence is called parser. For example, a group of words which is a series of words (phrases) and which word is the subject or objects of a verb and generates the grammar tree of it [Cor]. For example, first the statements are analysed, and their constituents are labelled with semantic roles, which have special importance to functionality: (1) Agent – Who performs? (2) Action

(the sentence's predicate or verb) – What is performed? (3) Object – On what objects it performed? (4) Instrument – How is it performed? (5) Temporal modifier – When is it performed? And (6) Adverbial modifier – In what conditions is it performed? [IRB14].

Therese are several methods for parsing process explained as follows [Sib+13]: (1) Top-Down Parsing is done by parsing a sentence starts from complete component (e.g., full sentence) to the smallest component (e.g., until we get the smallest component of the sentence that is the word). (2) Bottom-Up Parsing is done by taking one word at a time from a given sentence, to be assembled into larger components and done continuously until it forms sentences (i.e., began by a backward process with the sentence being parsed using the grammar rules until it reaches the start symbol).

2.4 Evaluation

In order to evaluate the contribution of this thesis (EvoSPL approach and friendlyMapper tool), we present an empirical case study of two parts: we conducted a case study following the guidelines, which are presented in [Lin+15] [Kit96], and we performed a validation part, where we show and discuss the obtained results. The concepts that are used in both parts are presented in this section.

Case study In the area of software engineering and information technology, there has been some interest in methodology of case studies, which are a standard method of empirical study in various sciences, such as medicine and psychology [Kit96]. Case study is a suitable research methodology for software engineering research, since it allows the effect of new methods and tools to be assessed in realistic situations [LSS05].

The case study methodology is well suited for many kinds of software engineering research, as the objects of study (e.g., new method) are hard to study in isolation. Case study was originally used primarily for exploratory and descriptive purposes [Fly06]. For evaluation, case studies that can provide valuable insights into why a new technology results in better products is needed [Kit96]. However, in this thesis we concentrate on presenting the 'which better' type case study, to provide valuable insights into why the EvoSPL approach results in better results than the baseline approach adopted in the automotive domain.

According to the guidelines of Runeson et al. [RH09], there are a set of major process steps (phases) to be undertaken, when conducting a case study. In the (1) design phase

objectives are defined and the case is planned. (2) Data collection is first planned with respect to data collection techniques and data sources, and then conducted in practice. Methods for data collection include, for example, interviews and observation. During the (3) data analysis phase, quantitative analysis (e.g., hypothesis testing) and qualitative analysis (e.g., chain of evidence) or one of them can be conducted. During the analysis it is important to maintain a chain of evidence from the findings to the original data. In the (4) reporting phase report is used to communicate findings of the study. The report should include sufficient data and examples to allow the reader to understand the chain of evidence. It is worth mentioning that a plan for a case study (design phase) at least contains the objective, case, theory, research questions, methods, and finally selection strategy elements [RH09].

Paired t-test The paired t-test, called sometimes the dependent sample t-test, is used to compare two means (i.e., means of two groups) that are from the same individual, object, or related units [Sta]. The two means can represent things, such as, a measurement taken under two different conditions (e.g., completing a test under a ”control” condition and an ”experimental” condition) or measurements taken from two groups of a subject or experimental unit (e.g., measuring performance of a team before and after adopting a new approach) [Ken].

Paired t-test can only be used when comparing the means of two groups, where the groups come from a single population (e.g. measuring before and after an experimental treatment). The paired t-test assumes your data are independent, (approximately) normally distributed, and have a similar amount of variance within each group being compared. You can test the difference between these two groups using a t-test [Scr].

Paired t-test is often used in hypothesis testing to determine whether a process or treatment actually has an effect on the population of interest (e.g., software developers), or whether two groups are different from one another. There are main steps in hypothesis testing, presented as follows [Scr].

1. State your research hypothesis as a null (Ho) and alternate (Ha) hypothesis.
2. Collect data in a way designed to test the hypothesis.
3. Perform an appropriate statistical test.
4. Decide whether the null hypothesis is supported or refused.
5. Present the findings in your results and discussion section.

Chapter 3

Related work

This chapter gives an overview of the related works and approaches for the evolution of SPLs, as focused by this thesis and its contributions. To give the reader some guidance, Fig. 3.1 shows areas of the related work aligned with the main phases of the EvoSPL approach.

3.1 SPLs evolution

As any piece of software, an SPL evolves over time. Its evolution process can be viewed either from an organizational perspective or from a process-oriented perspective. Practically, an SPL evolves whenever there are changes in (1) the requirements (i.e., changes to requirements lead to changes in the feature model), (2) the family structure, or (3) the technology being used. These new and changed requirements originate from several sources, such as customers using the products, future needs to be predicted by the company, and the introduction of new products into the SPL. Different approaches have been proposed for managing the artifacts of an SPL and some also address how evolution affects these artifacts [bastarrica2019software]. Existing mapping studies have focused on specific aspects of SPLs evolution and the nature of their processes, but there is no cohesive body of work that gives an overview of the area as a whole.

Marques et al. [SB99] provide answers to the research questions formulated to evaluate the approaches of an SPL evolution (using a set of 60 primary studies). The research was performed over workshops, conference and journal papers. Marques et al. show that the related area to the evolution of an SPL is maturing.

Challenges of an SPL adoption Marques et al. [SB99] reported open challenges that cause low adoption of SPL approaches in industry, such as: (1) the lack of research on

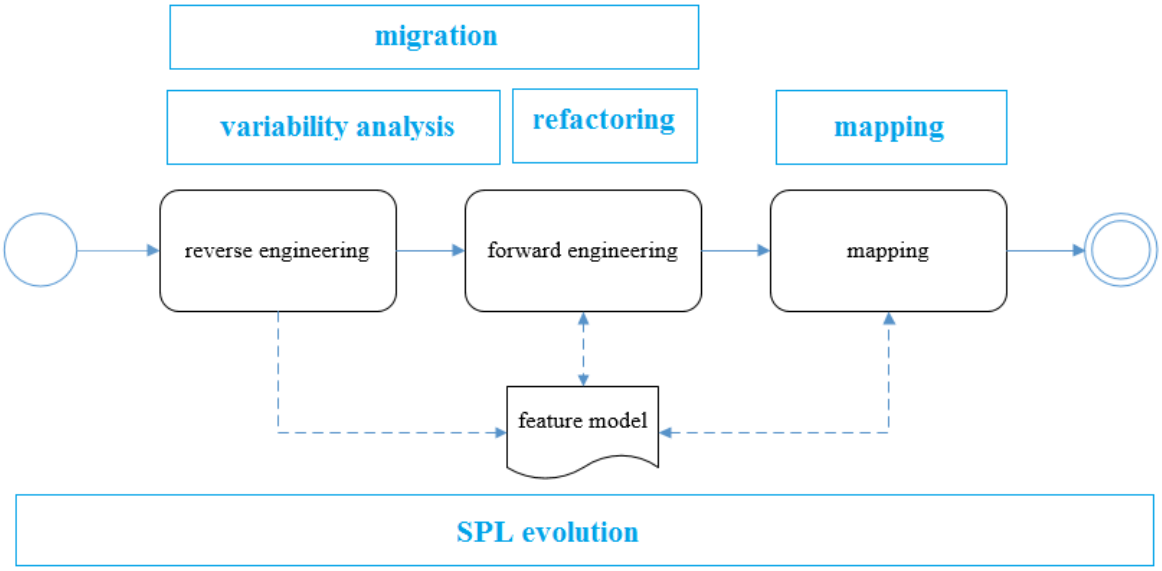


Figure 3.1: Areas of the related work aligned with the main phases of the EvoSPL approach.

traceability between the different artifacts of an SPL (this makes it difficult to completely assess the consequences of evolution on an SPL), and (2) the nonexistence of publicly available large industrial case studies that can be used by the industry as a starting point. Marques et al. believe that the SPL community needs to work together to improve the state of the art and to create methods and tools that support the evolution of SPLs in a more comparable manner. The EvoSPL approach presented in this thesis conforms to these conclusions and contributes a solution to tackle the open challenges presented by Marques. Hence, this thesis proposes a systematic approach called EvoSPL to manage the open challenges to the adoption of SPLs in the automotive industry.

Types of SPLs evolution Laguna and Crespo [LC13] and Montalvillo and Díaz [LC13] survey the different types of SPLs evolution. According to these two surveys, the first type depends on re-engineering of legacy systems (the existing products) into an SPL, as well as refactoring the existing products into the future SPL [LC13]. The second type depends on supporting the evolution of an SPL as a result of changes in requirements [MD16]. The EvoSPL approach proposes a re-engineering process to migrate the existing products into an SPL and then supports its evolution using an SPL refactoring.

SPL evolution process The software evolution process is quite challenging, since a balance between software quality and software structure must be maintained. Software

quality must be preserved while the software structure tends to upgrade over time. Botterweck and Pleuss [BP14] and Montalvillo and Díaz [MD16] identify the following challenges of SPLs evolution: (1) there are different types of artifacts defined at different levels of abstraction and with variability issues, (2) there is a high number of interdependencies between the artifacts, (3) an SPL usually has a longer lifespan than a single product, and finally, (4) an SPL is larger and more complex than its individual products. The approach presented in this thesis contributes to these open challenges and provides insights about the solutions.

Bayer et al. [Bay+99] developed the PuLSE methodology, a general framework for SPLE. The framework covers the evolution and maintenance of SPLs using PuLSE-EM (evolution and management). Based on the information provided as a result of other PuLSE components, PuLSE-EM provides an effective mechanism to propagate the change requests -e.g., product configuration history and product line architecture (PLA) history, which turn up during the construction and usage phases to the responsible components with adaptation. The PuLSE-EM and EvoSPL approaches both aim to enable the adoption of SPLs within an industrial context.

Abstraction level in the SPLs evolution SPLs evolution can be defined at different levels of abstraction, ranging from high-level artifacts, like requirement documents and architectural models, as presented by Garg et al. [Gar+03] to low-level artifacts, like code and test cases, as presented by Wijnstra [Wij03]. Like our approach, Pleuss et al. [Ple+12] present a model-driven approach to manage the evolution of an SPL at feature-level. Our approach defines the evolution of an SPL at a high-level of abstraction (i.e., at the requirement-level) and then propagates the changes to a lower one (i.e., the code implementation).

Safe evolution of SPLs Neves et al. [Nev+11] describe several safe evolution templates that the developer can use when working with SPLs. For this, Neves et al. rely on an SPL refined notion that preserves the original behavior of the products of a family. This notion helps to evolve an SPL safely, by simply improving its design or even by adding a new product while preserving the existing ones.

Neves et al. [Nev+11] present a study performed to evaluate the evolution of two SPLs using the templates. The results of the study show that the templates can address the modifications performed in safe evolution scenarios of the both SPLs. Similar to this

research work, the EvoSPL approach adopts a safe evolution scenario that can be used when evolving an SPL. This scenario depends on refactorings of SPLs. With the EvoSPL approach, the refactoring specification is aligned to the fundamental catalog of refactorings proposed by Alves et al. [Alv+06b]. The catalog includes the notations for specifying refactoring in a comprehensible manner. Furthermore, these notations are applicable to an FM to perform the transformation that improves and increases its configurability.

Model evolution and change Different models can be used to capture the changes between two model versions (e.g., two model variants) in terms of added, removed, and modified elements. In this context, Martinez et al. [Mar+14] contribute the MoVaC approach that compares a set of model variants and identifies both the commonality and variability in the form of features. Using a graphical representation, the MoVaC approach explicitly visualizes the common and variable features to users. The approach is validated on two case studies to demonstrate its applicability to any kind of EMF-based model (EMF - Eclipse Modeling Framework). In this thesis, the EvoSPL approach matches the requirements documents of two products at once and generates intermediate documents that identify the commonality and variability among them in terms of features. The EvoSPL approach adopts a visualization technique that expresses the commonality and variability of the generated documents in the form of colors and keywords.

SPLs evolution: an industrial experience Livengood [Liv11] describes an industrial experience to evolve an SPL of multifunction printers. The evolution is based on modifying the variability model in ways that alter the constraints and other relationships among variations. Livengood concludes that evaluating the impact of such changes on multifunction printers is difficult in practice and is an unsolved problem for the case described. Livengood believes that further research is required to solve such a problem. For that, he offers further research questions that may help in establishing further research. The EvoSPL approach contributes a solution for the research question (unsolved problem) presented in [Liv11]; “Can we improve traceability from the variability model to the development artifacts?” The EvoSPL approach supports a tool that automatically maps features belonging to the variability model (i.e., the current FM) to the respective code fragments, using the reference architecture as an intermediate artifact. Besides, the tool automatically updates the mapping links, upon the changes that affect an FM, to keep them consistent. The tool was evaluated using an industrial-sized products family and the result reveals that can be used successfully with a real products family.

The research work presented by Botterweck et al. [Bot+10] is motivated by requirements of the automotive industry. Hence, Botterweck et al. focus on evolving automotive parking assistants of a sample SPL. The authors developed an approach for feature-oriented modelling of SPLs evolution. Based on the sample SPL, the authors provide (1) a framework that includes modelling languages for SPLs evolution and (2) a catalog of evolution operators for the FMs. The presented approach provides the foundations for many evolution scenarios and consistency checking.

Botterweck et al. [Bot+10] specify the commonality and variability between the models over the time. For this purpose, the authors use a special kind of FMs, called EvoFM, to model the long-term evolution of an SPL. The FM of an SPL at a specific evolution step corresponds to a specific configuration of EvoFM. Consequently, the approach presents the evolution of the SPL in a compact view by using a sequence of EvoFM configurations over time.

In the literature, there are various approaches that support a tool for the SPLs evolution, e.g., [Rom+13] [Liu+07] [Dhu+08]. In practice, most of them address evolution based on the existing PLA of a products family [Gom06] [Gar+03] [Gar+03] [Ach+11], while other approaches design the PLA from scratch [SOB02] [SO01]. Some works address evolution in variability/feature model [Bot+11] [SSA14] [Pas+10] and code [Bot+11] [SSA14] [Pas+10]. Further approaches focus on pairs (or more) of artifacts that can evolve together [GF13] [ZCA17] [Gom13] [Men+09] [Wu+12].

Requirements evolution in SPLs Montalvillo et al. [MD16] present the results of a mapping study in the context of requirements evolution in SPLs, by answering the research questions formulated to this study. For the purpose of this work, the authors cover the evolution, which is triggered by requirement changes (not by bug fixing or refactoring). The authors wish such review can work as a starting point for practitioners to direct them in the field of an SPL evolution and to report their efforts in the whole field.

The evolution of requirements specification is a key activity of SPLs. This evolution is critical due to the necessity of dealing with common and variable requirements, not only for a single product but for the whole family of products. Oliveira et al. [OA15] propose the FeDRE approach, which provides support to evolve the requirements specification of an SPL in a systematic way.

Oliveira et al. also present the FeDRE2 approach, which provides support to evolve the requirements specification of SPL. The approach evolves the FM and the use case specification from an SPL according to the user's needs. It also provides detailed guidelines on how to evolve an FM and their related use case specifications. The approach is composed of three main tasks. The first task is to identify the evolutionary scenario for the requirements specification of an SPL. The second one is to perform the evolution for the requirements specification of the SPL. The final task is to update the Traceability Matrix, which stores the relationship among features and use cases. The approach has several steps to be executed to evolve the requirements of an SPL. Thus, through the proposed approach, the SPL requirements can be evolved in a safe way, keeping its traceability within the other artifacts. We believe (as authors of this thesis) FeDRE2 provides a solution to make it easier to achieve the intended evolution of requirements within SPLs.

Peng et al. [Wu+12] present a case study that can work as a starting point to understand the evolution of requirements in an SPL. The authors claim that software requirements are negotiated and may be guided by design of the existing software architecture, especially in the process of SPLs development. Using a case study in the industry, the authors propose a classification of requirements change (from the viewpoint of architectural impact of the SPL development in practice) and analyze the potential relationships between requirement change types and architecture evolution types. The authors find that requirement changes, in SPLs development, happen regularly and an architect may need more careful considerations before deciding to respond to it.

3.2 Migration towards SPLs

Today, many companies have highly diverse product families [BCH15]. These companies offer different products with different feature sets, to address more granular market segments and differentiate their products from the competitors' ones. Typically, the products of a family are somehow similar; hence they have a relatively large set of common functionalities (or features) and variants that make each product unique [LSR07]. This creates the opportunity to reuse and differentiate the products in a family. There are several approaches that describe processes for SPLs evolution. Numerous approaches focus on migration of the existing products towards the SPL (or extracting an SPL of the existing products).

Martinez et al. [Mar+15a] present an approach that automatically migrates the existing similar model variants into an SPL. The approach, named MoVa2PL, considers the identification of the commonality and variability in model variants. It provides end-to-end solutions that satisfy the required requirements when building a bottom-up approach that supports the SPLs adoption. The first requirement is feature location, which requires analyzing and comparing the existing model variants to identify commonality and variability in terms of features. The second is re-engineering, which requires a transformation of the model variants into an SPL. For instance, the variability model is created using information of the identified features and the discovered constraints. The EvoSPL approach presents a solution that supports the SPLs adoption in the automotive domain. In general, the approach migrates the existing products of a family into the resulting SPL. Firstly, it identifies features and the constraints among them to construct the current FM, which represents a given SPL and ensures valid product configurations for the family. Secondly, the EvoSPL approach evolves the existing SPL to encompass a (new) product.

Alves et al. [Alv+06b] present a method for extracting an SPL of the existing products and evolving the extracted SPL to encompass other products. The method systematically supports a strategy that uses the refactorings expressed in terms of simpler programming laws. Although the evaluation of this research work was conducted in the mobile game domain, the authors believe that the method is not only valid for mobile applications, but also other highly variant domains should benefit from the method. The EvoSPL approach applies a similar concept, since it supports the bootstrapping of existing products into an SPL and extending the bootstrapped SPL to encompass a new product. During this process, refactorings are performed to maintain the existing products and to add new products. The EvoSPL approach relies on the SPL refactoring that involves an FM defined by Alves et al. in [Alv+06b].

Martinez et al. [Mar+17] present a bottom-up technology for reuse, named BUT4Reuse. The technology is a generic and extensible open source framework that aims to support SPLs adoption. It can be used in different scenarios and with different types of software artifacts of a products family (e.g., requirements, models, or code). Also, the tool accepts to add different concrete techniques or algorithms for the relevant activities of SPLs adoption.

Martinez et al. [Mar+15b] believe that the practitioners lack end-to-end support, for bottom-up SPLs adoption, where a set of artifact variants already exists. To overcome

this challenge, Martinez et al. [Mar+15b] propose a generic and extensible framework called BUT4Reuse, and they assess its generic and extensible properties by implementing a variety of extensions. The authors introduce a framework for a bottom-up approach to SPLE with the aim to reduce the current high upfront investment required for a systematic reuse (end-to-end adaptor). The framework has been evaluated in the scenario of adopting an SPL approach of the existing Eclipse products family.

However, in this thesis, we believe that the automotive domain has specific requirements and it is suitable to dedicate this domain with an approach that takes its specific characteristics into consideration. Thus, the EvoSPL approach proposes a bottom-up approach with the objective to reduce the upfront investment required for a systematic reuse adoption in the automotive industry. Besides, the approach has been evaluated in the scenario of adopting an SPL approach of the existing products family in the automotive domain. Other approaches that aim to support the evolution of SPLs while focusing on migration are, e.g., [Fen+17] [Ass15], some of them describe industrial case studies, e.g., [Mus+18] [BLL08].

Managing variability in the FMs Metzger and Pohl [MP07] consider that a prerequisite to managing variability of SPLs is the explicit documentation of the variability. Thus, they present a tutorial on modeling and managing the variability of SPLs. The authors explain how to consistently handle the variability throughout the SPLE process and across all development artifacts. Moreover, the authors introduce a framework for SPLE. The EvoSPL approach conforms to this direction and explains how to consistently handle variability using a dedicated variability model (i.e., the current FM) throughout the SPLE process and across all development artifacts of a products family.

FMs and variability extraction Beuche [Beu15] presents a tutorial providing the required concepts on how to use FMs to manage the variability in SPLs. The tutorial presents an explanation about the role of FMs in SPLE, such as how FMs can be used to manage development and configuration of the products. Beuche and Papajewski [BPSP04] support the work appears in [Beu15] with a CONSUL tool. The tool supports the variability management that covers all the phases of domain analysis within almost all software development processes. The tool uses (extended) FMs as a main model for (1) describing the commonality and variability and (2) communication between developers of software artifacts to be managed. The tool has been proven to be an effective solution. Similar to this work, the EvoSPL approach manages variability in SPLs and its evolution using a sequence of the current FM with some level of automation.

Kramer et al. [KSRB13] believe that the variability knowledge expressed in an FM though may not be understandable to the end user. In practice, explanations have been shown to improve the intelligibility of software and improve user trust. Thus, this work presents how the variability of SPL based products can be explained to end-users using explanations composed from explanatory knowledge added to the enhanced FMs.

Martinez et al. [Mar+15a] address the problem of automating analysis of the existing similar model variants and migrating them into an SPL. In [Mar+15a] the authors describe the MoVa2PL approach to support a solution for the problem, by considering identification of the commonality and variability in the model variants, as well as extraction of the model-based SPL of features identified on these variants.

In this work, the authors address simultaneously in a single framework (an end-to-end solution and a bottom-up approach) both requirements (1. Feature identification and 2. Re-engineering) to extract an SPL of model variants. Similar to this work, the EvoSPL approach adopts a re-engineering process (end-to-end-solution) that supports analyzing and comparing existing products to identify the commonality and variability in terms of features. In contrast to Martinez's work, the EvoSPL approach uses requirement documents artifacts (i.e., that are developed using ad-hoc reuse techniques) to identify the commonality and variability among the products of a family by means of an FM. Moreover, as (new) products are added to the SPL, the FM is refined to reflect the existence of the new features. Al-Msie'Deen [AM14] proposes an automatic-based approach to identify features in the code and organize the identified and documented features (assign a name of the feature) into an FM.

Extract the variability from conditional compilation code In practice, conditional compilation directives (#ifdef blocks) are one of the most widely used mechanisms to implement variability (i.e., annotating feature code). Using conditional compilation directives, Couto et al. [CVF11] describe an experiment to extract eight complex and relevant features from the ArgoUML tool, in order to generate the SPL. Furthermore, Zhang and Becker [ZB12] present a model-based process that works on large scale SPLs. It automatically extracts both the variability and variability model from preprocessor code. The extraction process is based on the automatic parsing of the optional code implemented by conditional compilation directives. The variability tree indicates the dependencies between variations and modularizes these variations with a hierarchical structure. The extraction process is presented with concrete measurement results from the industrial case study.

SPL bootstrapping Regarding SPLs adoption in the marketplace, Li et al. [Li+16] are motivated from two perspectives. The first perspective is related to the need for research activities that adopt realistic case studies to evaluate (extractive) SPL adoption techniques. The second is related to the large and high scale study of reuse practice in an industry that is now growing among all others within the software engineering community. Therefore, Li et al. [Li+16] present a mining process of Android products with the objective of identifying families of apps in the market. The EvoSPL approach is motivated with the same perspectives presented in [Li+16]. Thus, our approach contributes an industry-based case study for evaluating SPLs adoption in the automotive industry. Also, our approach focuses on the practice of systematic software reuse in the industry.

Alves et al. [Alv+06b] and Tanhaei et al. [THMH16a] use SPLs refactoring to perform a process that involves bootstrapping existing products into an SPL and extending an existing SPL to encompass another product. In both cases, real case studies in the mobile game domain and the health domain are used to evaluate the work.

Reverse engineering of the FMs Al-Msie'Deen et al. [Ser+13] [urtadodocumenting] [UV] propose an automatic approach (1) to mine features from the code of a products family , (2) to document the mined feature implementations by giving them names and descriptions, and (3) to extract the FM of product configurations based on Formal Concept Analysis. To validate the approach, the authors apply it on several case studies. The evaluation results show the relevance and performance of the proposed approach, as most of the features and their associated constraints are correctly identified. Other approaches aim to reverse engineer an FM from other artifacts of a products family, such as architectural models [Ach+11], UML models [CD12], and informal products description [Dav+13].

3.3 Variability analysis

Generating variability from natural language requirements Schulze et al., [LSS17] present the results of a systematic literature review covering the approaches (that appear in 29 papers) to extract features and variability from NL requirements. The review allows the authors to make several observations. For instance, they observe that Software requirements specification (SRS) is frequently used as a main input for the feature and variability extraction from NL documents. However, the review reported that the

applicability of such documents in practice is questionable. Requirements specification document (RSD) is also has the same usage.

Bakar et al. [BKS15] conduct a systematic literature review on feature extraction from the requirements expressed in a NL. Both reviews reveal the same observation concerning the use of SRS as a main input for the feature and variability extraction. The EvoSPL approach also conforms to this observation, and it uses NL requirements documents (i.e., SRS) as an input to extract variability when conducting the case study at Bosch Car Multimedia company. Moreover, the EvoSPL approach contributes to evaluate the applicability of the SRS document in the extraction process, as questioned by Schulze et al. [LSS17].

Itzik et al. [IRB14] suggest a method to generate FMs from requirements or textual description of the products. The method considers stakeholders needs and preferences when generating the FM for the given tasks. In this work, the authors suggest an approach that measures the semantic similarity, extracts variability, and automatically generates FMs (that represent structural or functional perspectives of the given context). The EvoSPL approach adopts some of the concepts presented in [IRB14] to generate the current FM from the requirements documents of a given products family. Our approach analyses and pares the variability terms of requirements, and it adopts the semantic roles that have special importance to the automotive industry to identify features.

As FM construction can be tedious and time-consuming, many researchers have previously developed techniques to extract FMs from sets of formally specified software requirements specifications for families of existing products. For instance, Davril et al. [Dav+13] present a novel approach to automate the constructing of an FM from a set of available and informal descriptions.

Acher et al. [Ach+12] propose a semi-automated tool-supported process to extract the variability of a family of products. An automated technique to synthesize FMs based on merging a set of products descriptions was developed. Acher et al. aim at easing the transition from product descriptions expressed in a tabular format to FMs, accurately representing them. The authors reported preliminary experiments based on publicly available data that appears in [HTM09].

Fine-grained and coarse-grained granularity Fine-grained granularity, like adding a statement in the middle of a method, normally pollutes the code with annotations.

Typically, an SPL implementation supports features with coarse-grained granularity, like the ability to add entire methods. Furthermore, some existing SPLs could benefit from fine-grained extensions to reduce code replication or improve readability. Though many SPLs can and have been implemented with the coarse-grained granularity of existing approaches. However, the fine-grained granularity extensions are essential, when extracting the features of a products family. Thus, Kästner et al. [KAK08] analyze the effects of feature granularity in SPLs and present a tool, called CIDE, which allows features to implement coarse-grained and fine-grained extensions in a concise way.

Manage variability using the background colors The CIDE [KAK08] enhances feature management with a tool support. It is based on preprocessor semantics, but it uses the background colors instead of code statements, and it provides the possibility to hide features to avoid code pollution. For instance, the tool indicates the features with different background colors. Thus, developers can directly recognize whether the code fragment is associated with a specific feature. In case of a code fragment is associated with multiple features (i.e., nested preprocessor statements), it mixes the according background colors. For example, it mixes the red and blue colors in purple. Using two case studies, the authors show that CIDE simplifies SPLs development compared to the traditional approaches.

In EvoSPL, a feature specifications and implementations are allowed in the requirements document and code, respectively, with a fine-grained and coarse-grained granularity. Besides, our approach not only provides a visualization technique that identifies the variability (common/variable requirements and features) of a products family with colors, but it also relates a feature to the corresponding code fragments, considering a coarse-grained and fine-grained implementation of the existing code.

Feature identification Ziadi et al. [Zia+12] propose an approach to automate feature identification from the code of a set of products. The proposed approach is composed of three steps, as follows. The first step retrieves a model from the code of each product and decomposes the model into a set of atomic pieces. The second step uses an algorithm to produce the feature candidates. The third step manually produces the final set of features of the SPL, which will serve as the basis to build an FM. After conducting the experiments to evaluate the approach, the authors suggested that the approach is promising but requires more work. For instance, he recommended proposing guidelines that support the building of FMs. The EvoSPL approach presented in this thesis proposes a method (called FIM) that identifies feature candidates from the requirements document of the products, and then the approach uses them as a base to construct the current FM.

Name suggestions during the feature identification Al-Msie'Deen [AM14] exploits the feature related code and use cases in documenting the identified feature, by providing a name and description. The feature related code and use cases contain together the external functionalities with textual descriptions of these functionalities.

Martinez et al. [Mar+16b] propose a process, named VariClouds that helps domain experts to assign a name for a feature during the feature identification. The VariClouds can be used to understand semantics behind the identified different blocks during analysis of a products family. The process, which works with different types of artifacts (i.e., models or code), is based on interactive word cloud visualizations, to provide name suggestions for these blocks. The VariClouds defines two phases for the domain experts. The first phase prepares word clouds (created using the words from any set of elements), and the second phase obtains the blocks (using the block identification algorithm) and lets the domain experts use the interactive word clouds to name each block.

The EvoSPL approach proposes a method that suggests a name during feature identification using the text-based parsing and NL technique. Our approach (1) analyses a requirement statement and identifies the main terms that have an important semantics of the automotive domain, (2) performs normalization on the parsed terms of each statement, and (3) assigns one of these terms to the feature. The software engineers can decide about the name of the feature name, to either accept it or reject the name. In case of rejection, the software engineers propose a proper name and then the EvoSPL approach investigates the consistency between the proposed name and the requirement statement to decide on the final name of a feature. This reworked as many times as possible to resolve the conflicts concerning the feature name until the proposed name turned into an agreement.

Managing variability on the code-level Mengi et al. [Men+09] believe that the current approaches for modeling and managing variability on code-level do not consider specific requirements of the automotive domain. For that, Mengi et al. describe an approach that links the variability model with the code. The basic idea behind the work is to shift the work steps into the variability model, to model, manage variability, and implement their variables in the code.

The EvoSPL approach presented in this thesis complies with the work presented in [Men+09] and contributes a solution to tackle its future work. The EvoSPL approach

investigates the earlier artifacts of the development process, including requirements document and shared software architecture of a products family to explore the variability.

Managing variability of the SPL architecture According to Garage et al. [Gar+03], there is a lack of support for managing the evolution of SPLs architecture. Hence, they provide a tool to tackle this problem. The presented tool environment Ménage provides an architect with the (visual) ability to specify and evolve the architecture of an SPL as new product architectures are added or existing product architectures are modified or removed. The EvoSPL approach manages variability in a setting where the reference architecture is shared for the products of a family. The EvoSPL uses the reference architecture to relate explicitly each feature of the current FM to its implementation code fragments in the code.

Model-based evolution At the SPL-level, changes to requirements lead to changes in the variability model. Many models are used to manage the evolution of SPLs. Feature modeling is the most commonly used one, like the evolutionary FM presented by Pleuss et al. in [Ple+12]. UML diagrams, like class diagrams, component diagrams, state collaboration diagrams [Gom13], use case diagrams [OA15] and state diagrams and sequence diagrams [Liu+07], are also used. Other UML diagrams such as are used as well. The EvoSPL approach uses the current FM, to keep track of similarities and differences of the SPL under consideration and to deal with modifications that are made to the SPL over time.

Understanding feature evolution As features represent an SPL in a way that is meaningful to different stakeholders, Xue et al. [XXJ10] suggest that understanding of how features evolved in a products family is a prerequisite to transition from ad-hoc to systematic reuse. Hence, Xue et al. propose a method that assists the analysts in detecting changes that affect features of the product during its evolution. Xue et al. apply a model differencing algorithm to identify the evolutionary changes that occurred to features of the multiple products of a family. The authors evaluate the effectiveness of the approach on a family of medium-size financial systems. The evaluation demonstrates that the approach yields good results. Using the EvoSPL approach, the evolution is modeled as a sequence of changes that are applied to the current FM, which present a conceptual modelling of the evolution and changes in SPLs.

Modelling evolution of SPLs: change operators The change operators performed on variability models (e.g., FMs) is a basic concept to specify changes in an SPL. Botterweck

et al. [Bot+10] provide a feature-based approach to model the variability over time and a catalogue of change operators on FMs. Their approach models the overall evolution of an SPL focusing on variability in time. The basic idea behind this approach is to use a special kind of FM called EvoFM. Each EvoFM at a certain point of time can be described as a composition of fragments. The changes within fragments (e.g., changing a feature from common to optional or adding a new feature) are specified by change operators associated with the fragment. Hence, the overall evolution of an FM can be represented as a sequence of EvoFM configurations.

The research presented by Botterweck et al. is motivated by characteristics of the automotive industry. The authors focus on SPLs of embedded systems, in particular control systems in an automotive environment (i.e., the sample SPL of evolving automotive parking assistants). The authors assume that such systems are created with approaches from Model-based Engineering of Embedded Systems and the implementation is described in a domain-specific (modelling) language for embedded systems, such as Simulink.

Both Xue et al. [XXJ10] and Botterweck et al. [Bot+10] propose a set of evolution operators (feature changes) to describe all the possible changes on an FM (e.g., add, remove, and modify). The EvoSPL approach presented in this thesis is aligned with the works presented in [Bot+10] [XXJ10]. Hence, our approach expresses various change operators that are applicable to the current FM and the existing SPL. The change operators are used in the process of our approach to refine the current FM twice: (1) when bootstrapping the remaining products of a family into the initial SPL and (2) when extending the bootstrapped SPL to encompass a new product. The EvoSPL approach is motivated by the requirements of the automotive industry, in particular the sensors of control systems. The basic idea of the EvoSPL approach is to capture the variability and evolution of the automotive domain families at the requirements-level, in forms of a sequence of the current FMs.

Flexible tool-based approach with industrial examples Research on SPLs often neglects the handling of documents such as contracts, technical documents, or user manuals. In practice, stakeholders often need to adapt these documents in SPLs. To address this issue, Rabiser et al. [Rab+10] implement the DOPLER approach and tooling suites to model the variability of documents in SPLE and to present variability to users in product derivation.

The DOPLER approach comprises four steps: (1) elicit and analyze the variability in documents, (2) create or adapt variability models, (3) choose or develop a variability mechanism and a corresponding generator for domain-specific document formats, and (4) generate the documents with variability information. Rabiser et al. [Rab+10] applied successfully the described approach (DOPLER and DocBook tool suites) on two industrial SPLs with different level of maturity. Both industrial partners plan to integrate the DOPLER approach in their process.

Managing variability and traceability via specifications Ghanam and Maurer [GM09] describe a case where they handled variability in the domain of intelligent home systems to satisfy a range of requirements for the industrial partner. The work delves into how the variability and traceability of requirements can be managed via executable specifications. This approach has been evaluated through a case study provided initial insights on its feasibility and usefulness. Ghanam and Maurer present three main factors at play in SPLE: 1) commonality and variability management of requirements; 2) traceability of commonality and variability from the requirements to the code; and 3) managing and tracking reuse of code across members of a products family, usually driven by the previous two factors. The EvoSPL approach presented in this thesis proposes a re-engineering approach that consists of three main phases: 1) reverse engineering; 2) forwarded engineering; and 3) mapping. The phases explore into the factors to accumulate a solution that satisfies all of them (factors).

Variability-modelling practices in the industrial SPLs Existing studies regarding an SPL practice mostly describe in general with little focus on variability modelling. Nair [Nai13] conducts a qualitative study on practices of the variability-modelling in the medium-and large-scale companies. The study uses two empirical methods: surveys and interviews. Nair investigates the practices and experiences of the companies under the consideration, regarding variability-modelling, with the aim to gather information on methods, strategies, and tools used to create and manage the variability models. Additionally, he aims to gather information on the perceived value, characteristics and challenges of the variability modelling. The results in [Nai13] reveal that variability models are often created by re-engineering existing products into an SPL. In addition, the results show that the challenges were related mainly to the visualization and evolution of the variability models and dependency management. Nair concludes that the adoption of SPLs and variability modelling have forced developers to think in terms of SPLs scenario rather than a product-based scenario.

Clone detection (identify features in forks) In practice, code forking (or clone-and-own) is a commonly used approach in the industry; where a new product is created by cloning the existing ones. Several studies have investigated the practices of forking in open source and industrial development of SPLs [Dub+13] [NM11] [SSW15]. Generally, these studies reveal the discussed problems, but did not provide any solutions. Zhou et al. [Zho+18] and Rubin et al. [Rub+12] propose a solution that mitigating the disadvantages of the forking mechanism while exploiting the advantages. They propose an approach that identifies the changes and feature implementation in forks. Zhou et al. [Zho+18] introduce an INFOX approach that automatically identifies and labels features within the larger change of a fork.

Tairas et al. [TGB06] describe a stand-alone clone detection tool, called CloneDR that is integrated into Eclipse. The tool can identify the sections of a code that are duplicated in the program and shows the visualization of clone detection results. Besides, the tool can help users to identify certain characteristics of the clones, by graphically isolating clone groups across a list of source files. The EvoSPL approach presented in this thesis uses Beyond Compare tool, to quickly and easily compare folders and text-based files of the code, which helps to identify commonality and variability among the code modules of a products family. Furthermore, EvoSPL uses the tool to compare text-based files of the code artifact of two products and to detect clones between them. However, managing clones have been addressed in the context of FMs for the first time in [Rab+16].

3.4 Refactoring

In practice, adopting SPLs does not start from scratch, but rather start with bootstrapping existing products into an SPL and extending the SPL to encompass another (new) product. One way to achieve this is to use refactoring. Alves et al. [Alv+06b] propose a catalog of sound refactoring for FMs, to guarantee configuration improvement. Each refactoring consists of two temples (left-hand side and right-hand side template). The catalog applies refactoring whenever the left-hand side template is matched by a given FM. The catalog supports a set of operations (e.g., a change made to the FM) to be applied to the FM that preserve (maintain or increase) the products of an SPL unchanged. The authors evaluate the proposed refactoring notation in a real case study in the mobile domain.

Usually, an FM is used to keep track of similarities and differences among products of the SPL. Thus, Tanhaei et al. [THMH16a] propose a framework for refactoring SPLs, which helps keep an SPL consistent with its related FM. Tanhaei et al. introduce a set of refactoring patterns that are applicable to the FM and the SPL (i.e., resulting in features being added to or eliminated from the SPL, like adding a feature to the FM). Furthermore, the authors introduce the required algorithms to implement a tool that performs refactoring on the FMs. The aim of this framework is to use the proposed refactoring patterns in modifying the artifacts of an SPL (as the reference architecture, FM, design, and code) in a way that keeps all the product configurations valid. The authors evaluate the applicability of the framework presented in this work, using a real-world case study (real-word SPL).

In the case study, the authors investigate the 'Medio SPL' to find the refactoring opportunities that can be applied to this SPL. The 'Medio SPL' consists of four products, three products are derived from the existing versions (products) of the SPL and the fourth one is an upcoming product (a new product) schedule for being developed in the near future. Using these products, the authors used two teams in evaluating the framework. The first team uses the framework presented in [THMH16a] to perform refactoring on the SPL. The second team is an external team that does not use the framework to find and perform refactoring on the SPL (they suggested refactoring to 'Medio SPL'). The authors compared the performance of the two teams. The results of the case study reveal that the framework can find refactoring opportunities in the FM that the second team (the team they did not use the framework to perform refactoring) cannot find. Finally, the authors concluded the benefits of using their framework in refactoring the 'Medio SPL'.

Tanhaei et al. in [THMH16b] propose a safe framework that supports refactoring of an FM. The authors define some of the important refactoring rules on the FM and provide tools that enable the users to add new rules. The authors propose a way to assess the correctness of an FM refactorings. The framework converts every FM instance to an Alloy model, and then uses Alloy Analyzer to evaluate the correctness of the edits performed on it. Calheiros et al. [Cal+07] present a code refactoring tool, named FLiPEx, which can be used to extract products in the context of developing mobile game SPLs. Similar to [Alv+06b] [THMH16a], Barba proposes a notion and catalogue of refactoring for SPLs (that works for any FM). The catalogue contains transformation templates that separately deal with FMs. The notation guides and improves the safety of the derivation and evolution of SPL processes.

3.5 Mapping in the context of SPLs

SPLE and SPL evolution require mapping features to their implementation. This problem is addressed by the feature location [Mar+16a]. In the context of SPLs, the mapping between features and implementation artifacts can be established by a kind of tracing links. Traceability relates the artifacts relevant to variability management, like mapping between an FM and the code. Vale et al. [Val+17] perform a study to understand the state-of-the-art in traceability of SPLs, which leads to indicating areas for further research. The authors concluded that (1) SPL traceability is maturing and requires further investigation, and (2) defining and maintaining of the trace links require much effort, so there is a need for tool support.

Tool support Zheng et al. [ZCA17] propose an approach to address feature-architecture mapping and architecture-code mapping. The approach can trace a feature of an SPL (modeled on this work using a feature tree) and the code, and it can automatically update the architecture and the code to maintain their conformance whenever feature changes occur. The authors develop an Eclipse-based toolset named xLineMapper to support the approach presented in [ZCA17]. The tool set includes (1) a graphical modeling tool, (2) a code generator, (3) an annotation processor, and a (4) code visualizer. The approach presented in [ZCA17] may require several changes to be made to the code, to adopt the approach. In contrast, the EvoSPL approach presented in this thesis uses feature mapping activity, to relate each feature of the current FM using the reference architecture as a centric point. In contrast to the work in [ZCA17], our approach does not require upfront effort from the software engineers to adopt the steps of the approach. The feature mapping activity is supported with friendlyMapper tool that automate the mapping process between features and code fragments. Besides, it automatically updates the mapping whenever changes occur to the current FM.

Meinicke et al. [Mei+16] provides an Eclipse-based tool, called FeatureIDE. The work aims to tackle multiple challenges with preprocessors, such as code comprehension, feature traceability, separation of concerns, and program analysis. With FeatureIDE, instead of focusing on one particular preprocessor (CPP), the authors provide tool support, which can easily be adopted for further preprocessors (Antenna, and Munge).The mapping phase of the EvoSPL approach is supported with a tool called friendlyMapper that reads XML file and imports it to the tool environment. The tool reads the features from the XML file and establishes the tracing links that relate each feature to its implementation code

fragments. However, all the FMs presented in the thesis are designed with FeaturIDE editor [Kas+09] [Bot+11] [KA11].

Feature location According to Martinez et al. [Mar+16a], feature location for software families is a research field that is becoming more mature with a high diversity of techniques. Martinez et al. propose a publicly available framework and benchmark, called EFLBench that provides a common ground for this field. The benchmark is based on the Eclipse releases and it is designed to support research on software reuse in the context of SPLs. The EFLBench is publicly available and supports all the tasks for techniques of the feature location, like integration, benchmark construction and benchmark usage. However, the authors have shown examples of usage of the benchmark with the Eclipse products family for analyzing four different feature location techniques, and they have discussed the evaluation of one of the feature location techniques using (randomly generated) sets of Eclipse products. In this direction, the EvoSPL approach presents a process that satisfied the requirements of feature location identified by Martinez et al.

Martinez et al. [Mar+15b] illustrate a methodology to describe the extraction of an SPL of the existing products. The method consists of the following numbered steps: (i) preparation of artifact of the products, (ii) feature identification and location, (iii) feature model creation, and (iv) reusable artifact extraction. The approach presented in [Mar+15b] does not assume a complete upfront knowledge of the existing features throughout the artifact of the products. In this work, the authors aim to have an explicit list of the features within the scope of the family of products. For that, they adopt the work in [Mar+14] to automatically identify the implementation blocks that represent a feature and assign them a proper name (feature name), using information retrieval techniques [Mar+16b].

In contrast to the default assumption in [Mar+15b], regarding the non-availability of features, the authors explain another scenario. As if the features are known in advance, and their presence and absence in the artifacts is known, feature location can be performed directly after creating the feature list. The feature list model contains a list of features with the reference to the artifact variants that have or implement each feature. In the written works, there are many works of approaching feature location, e.g., [Ser+13] [Ra 3] [XXJ12].

Feature location techniques According to Ra'Fat Al-Msie'Deen [AM14], feature location techniques aim at locating artifacts of a products family that implement a specific

feature. Al-Msie'Deen contributes a new approach to mine (identify/find) features from the (object-oriented) code of a set of a products family. The approach is based on three techniques to do the work: Formal Concept Analysis, Latent Semantic Indexing, and analysis of structural code dependencies.

The mappings between use case diagrams and FMs Inside the SPL community, features have been widely used to model variability. From a user perspective, use cases are also widely used to model the functionality of systems. Significant work has been done to relate variability and use case models. Bragança and Machado [BM07] present an approach that maps use cases to features in a formal way and describe a possible implementation of the approach (to automate the transformation from UML use case to FMs). The contribution of this work was inspired in the works appeared in [GFd98] [Gom05].

The issue of relating use cases and features is not new. Grisset al. [GFd98] propose an approach to extract functional features from the domain use case model. They also explain how the structure of the FM can be created according to the structure of the use case model (i.e., by using the «include» and «extend» relationships). There are others similar approaches that relate use cases and features, e.g., [EBB05].

Integrating feature and architecture Models Janota et al. [JB08] provide a formal foundation that integrating the FM and the architecture model of a family of products (i.e., by providing a formalization of dependencies between features and components). Such integration between the models helps to tackle the risk of inconsistencies between them (i.e., the feature model might allow feature configurations that are not realizable by the architecture) and offers a better understanding of the modeled concepts. For instance, the approach offers support to formalize models for features and their implementations, which can be utilized to provide automated feedback (i.e., additional information) to the software engineer who uses such models.

Tracing variability from features to artifacts of the SPL Díaz et al. [DPG15] focus on traceability between artifacts resulting from the domain analysis and architecture of an SPL (i.e., the traceability between the FM and the PLA model). Díaz et al. present a solution for tracing the variability from the features to the architecture of an SPL (PLA) taking variations in components and variations inside components of the architecture (both external and internal architecture variability) into account. The solution is supported by the Featured-PLA model framework, which has been deployed in an industrial project on Smart Grid [AW05].

The mapping phase of the EvoSPL approach provides a tool, called friendlyMapper. The tool relates each feature of the current FM to its implementation code fragments in the code using software architecture as a bridge. Besides, it constructs the traceability tree that maintains the tracing links between feature and code fragments, whenever feature changes occur. We evaluated the tool as a part of a case study conducted at Bosch Car Multimedia company. The results reveal that our approach, including friendlyMapper, is both applicable and capable to support the evolution of product families in the automotive industry.

Chapter 4

EvoSPL approach

4.1 Approach overview

This chapter presents the approach proposed by this thesis and explains its process and main phases. To tackle the challenges of evolving an SPL that takes the existing products of a family into account (re-engineering), this thesis proposes an evolution-based approach named EvoSPL. The approach considers a process which evolves an SPL from the existing products of a family and focuses on the migration of the existing artifacts (namely requirements document). The basic idea of our approach is to model an SPL and its evolution, by focusing on and describing the commonality and variability in the requirement documents of a products family. Thus, in EvoSPL, evolution of an SPL is represented as a sequence of FMs at different points. At each point, the approach refines an FM with the features of a (new) product.

The approach (i) delivers detailed similarity and difference analysis between the products of a family, (ii) detects commonality and variability among the products of a family, and (iii) supports a systematic migration process of a products family. Consequently, this helps to manage variability and to consistently evolve the SPL under consideration. The information related to the process, activities, techniques, and contributions of the EvoSPL approach is detailed in the upcoming chapters.

4.2 The approach main phases

Fig. 4.1 depicts the EvoSPL approach at a relatively high-level of abstraction. The three-phase approach aims to adopt the SPL in a given context. The phases are reverse engineering, forward engineering, and mapping. Each phase of the approach has specific activities (not shown in Fig. 4.1) to be performed according to the process of the EvoSPL

approach. The FM works as a common model that is shared among all the phases. Phase 1 derives it, phase 2 upgrades and refines it, and phase 3 maps it to code.

The first phase (reverse engineering) uses requirements documents of two products of the same family to derive an FM, in order to support the explicit variability management of an SPL. This phase applies a difference analysis that compares the requirements document of each product against each other to detect similarities and differences between them. Then, this phase applies the variability analysis and feature model synthesis to generate the current FM, which is an FM that has been derived from two products and reflects the initial SPL and partially the future SPL.

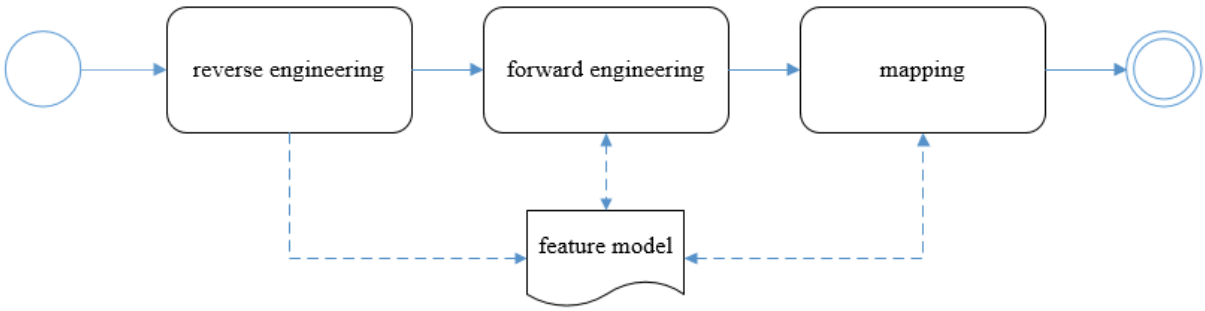


Figure 4.1: EvoSPL approach main phases.

The second phase (forward engineering) starts with the bootstrapping activity. This activity uses the current FM and requirements document of another product, to bootstrap (one by one) the remaining products (the ones that were not used in the first phase) into the initial SPL. The bootstrapping activity initially uses the current FM that has been derived from two products and refines the current FM when required (e.g., upon adding a product to the initial SPL under the consecration). Besides dealing with bootstrapping, this phase also needs to deal with the evolution of an SPL. For that, the evolution activity evolves the bootstrapped SPL with a new product according to new user's needs. Both activities refine the current FM in accordance with features of a (new) product.

The first two phases of the EvoSPL approach perform a semi-manually activities with tools support that have been adopted by other researchers. The final phase (mapping) of the approach supports an automated mapping for artifacts of the SPL (i.e., tool support). In order to facilitate the adoption of SPLs that reflect the need of the customers and perspective of the software team, our approach allows a manual intervention of the software engineers in performing the EvoSPL activities.

4.3 The EvoSPL approach

This section describes the main activities, inputs, outputs of the EvoSPL approach. The approach aims to provide a consistent variability management and a variability-aware refactoring technique towards moving into an SPL. The approach process comprises the main phases and sub-phases of the approach. In addition to pre- and mid- processing activities to initiate and integrate the main phases.

4.3.1 Definitions

The following are definitions of the main terms that are used by the EvoSPL approach.

Feature The definition presented here for the feature introduces a flexible support of the approach to generate the FM in a given case. A feature is a unit of functionality that satisfies a set of requirements, including components, hardware, technologies, services, or other functionalities that are provided by the products of a family to satisfy customer needs [BKS15]. Our approach assumes that features represent decisions (variations) and provide configuration options (the products of a family). When generating a products family, a feature can be common (shared by all the products of a family) or optional (differ between the products of a family) [Dav+13]. In consequence, a set of products generated from a systematic modeling of features is called an SPL.

Products Family A products family refers to a set of similar products that are derived from a common platform and have specific features/functionalities to meet particular customer requirements that target a certain market segment [DJT01]. All products share some common structures, technologies, and /or components, which form the platform of the products family.

Product A product is an individual member within a products family that is developed to address a specific set of customer needs within the market segment. A product has a collection of artifacts that implement the features of a single software product. These artifacts are copied (or developed) and modified (or customized) to satisfy specific requirements.

When applying the EvoSPL approach, we assume that a product is developed by cloning either the initial release product or another product in the family that is more suitable, and by adapting it to satisfy specific customer needs [DJT01]. Practically, the initial

release product is cloned from the platform artifacts, including requirements document and code, then it is customized (the customization technique is not mattered to our research). Thus, we guarantee the minimum required similarities among the products of a family to apply our approach.

Initial release product The initial release product is a software product of the family that is cloned from the platform artifacts and customized to satisfy customer needs. Normally, the initial release is called the first release product and developed to be used by the first customer.

Variation In EvoSPL, variation represents a place in which variability occurs among the products of a family. Typically, the place can be a text fragment in the statement of the requirements document, a component of the reference architecture, or a fragment of the code that implements variability of a products family.

Variant The term variant appears in our approach to represent available alternatives for a variation place. Each variant specifies or implements one or more statements of the requirements document, component of the reference architecture or code fragments of the code of a products family.

Variations and variants are used to define the variability of an SPL. Thus, it is essential to be able to identify variations and variants in a systematic manner. In the following, we provide an example that explains both concepts from the customer and the model view. The variability in 'connectivity', from the custom view shown on the left of Fig. 4.2, has two methods ('Bluetooth', 'WI-FI'). A mobile phone company wants to develop mobile phones in different connectivity methods; therefore, a variation 'connectivity' is defined in the model view. The company develops mobile phones with 'Bluetooth' and 'WI-FI' connectivity methods, therefore the variants 'Bluetooth' and 'WI-FI' are defined in the model view on the right of Fig. 4.2.

SPL terms As shown in Fig. 4.3, the initial SPL term refers to the SPL that has been extracted using two products of a family (P1 and P2), in reverse engineering phase. The foreword engineering phase involves bootstrapping the remaining products of a family (P3) into the initial SPL, to establish the bootstrapped SPL, and extending the bootstrapped SPL to encompass new product (NewP), to deliver the resulting SPL. The resulting SPL can be extended to encompass another product when required. The resulting SPL is

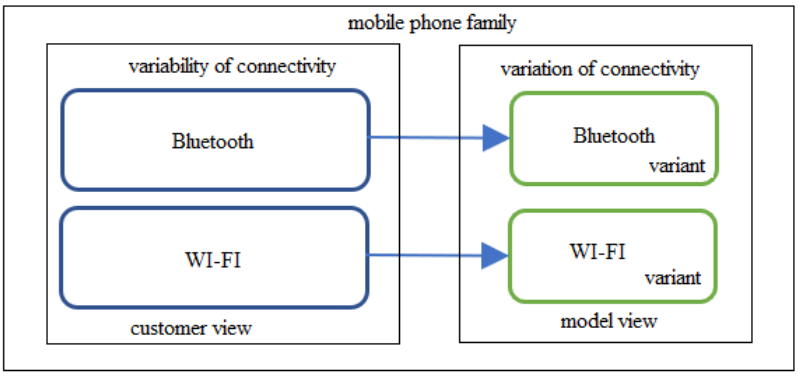


Figure 4.2: Relationship between variability in the customer view and in the model view.

used in the mapping phase to map features of the current FM that models this SPL to the code, to preserve the consistency among them, using the reference architecture as an intermediate artifact.

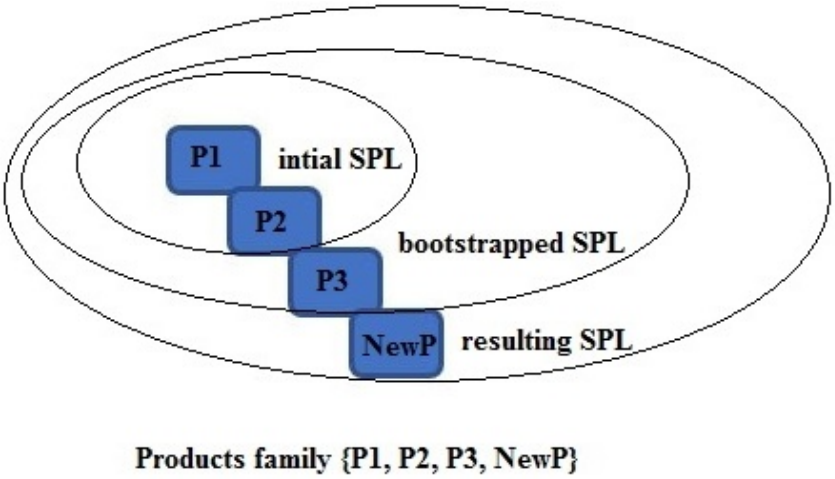


Figure 4.3: The SPL terms used by the EvoSPL approach.

4.3.2 Inputs and outputs of the EvoSPL approach

The following artifacts are the input of the EvoSPL approach.

Requirements Document A requirements document is an artifact containing all known requirements to a certain product of a family. Our approach uses requirements documents that are documented using NL text (e.g., English) and organized according to a previously established format (with predefined sections). It is written to allow customers to understand what the product will do, in order to later allow the software engineers to use their expertise to provide a solution to the requirements. An SRS and RSD are

the most popular documents written to specify a software product from a user's and producer's point-of-view [BKS15]. Some additional information in a tabular format that may appear in the requirement document is accepted by the EvoSPL approach. However, the requirements models (e.g., UML diagrams and flowcharts) are not accepted by our approach.

Code The implementation code is an artifact organized in modules (packages). Each module consists of one or more source files, which have been evolved over the years and implemented with C-C++ language. We assume that the code uses the following mechanisms to implement and realize variability.

- C-C++ preprocessors directives, like file inclusion, macros definition (#define and #undef), and conditional compilation with #ifdef blocks. For the rest of this thesis and for brevity, we use #ifdef blocks to also includes the related directives of #ifdef, #ifndef, #if, #endif, #else and #elif.
- Conditional execution, like if-else blocks and switch blocks.

Reference architecture The reference architecture of a products family is taken into consideration, since it includes the variability documented in the variability model (i.e., the current FM). Typically, this artifact represents a core architecture of the resulting SPL, which is originally shared among the individual products of a family. The reference architecture, in our approach, consists of common and variable software components (corresponding to software modules), for example, variable architecture components reflect the differences among the products of a family. These differences (variations) appear in both the design and implementation respectively.

With respect to the outputs of the approach, different types of artifacts are produced, including.

Single master document A single master document (SMD) is an artifact that consolidates the variability information of a products family into a single document.

Features list The features list (FL) is an artifact that contains a list of features of the product(s) with the references to the requirement statements that have/implement each feature, the variability-pattern of each feature, and the relationships and constraints among the features.

The current FM The current FM is a primary artifact to model the initial SPL and the bootstrapped SPL and its evolution at a specific point, to support the delivery of the final SPL. Each point represents an evolution step of the SPL, using the current FM, beyond evolving the SPL with a (new) product.

Traceability tree The traceability tree is an artifact that stores the tracing links between the current FM and code. Each tracing link maps a feature to the corresponding code fragments that implement this feature, which enables the current FM and code to be linked and to be evolved consistently.

4.4 Requirement engineering in the automotive domain

Typically, requirement engineers in the automotive domain express the requirements at a detailed and technical level. Requirements at the different levels of abstraction are ranging from goals to detailed technical requirements, including hardware details that need to be included in the requirements document [Bra+14]. High-level requirements (e.g., RSD) provide a justification for detailed requirements and support the understanding of the requirements. Low-level requirements (e.g., a use case description) are detailed, and they need to provide enough information for implementing the system correctly.

Automotive development is in general too complex to be managed by requirements documents written in NL (e.g., textual requirements) alone [WW02]. Definitely, this does not mean that textual requirements are unnecessary, but at the same time using just the textual requirements in automotive development is clearly insufficient. However, due to the large size, variety, and complexity of the automotive development, a good model (notation, method, and tooling) is necessary to describe the automotive features and variants to handle variability at a higher abstraction level.

4.5 The primary input artifact of the EvoSPL approach

The adoption of SPLE to support the systematic reuse of software-related artifacts of a products family is challenging, time-consuming and error-prone [IRBW15]. Considering that requirements drive many development methods and activities, this thesis introduces an approach that uses requirements, as presented in textual requirements, to analyze the variability of a products family. Typically, requirements documents that are written in

NL may be structured in various forms. According to a review presented in [LSS17], SRS is frequently used as an input for feature and variability extraction. The structure of such documents (is designed by other research as RSD) may contain valuable information, such as for grouping features or establishing parent-child relationships. Due to considerable progress in NLP, we use requirements documents, typically in the SRS format, as a primary artifact for feature and variability extraction (see chapter 5). A variety of information, including commonality and variability, can be extracted from such requirements.

Existing approaches focus mainly on code to extract the variability of a products family [LSS17]. In this work, we decide that an effort may be required for using requirements documents to extract variability, due to the missing traceability links from the code.

Software developers typically focus on requirements engineering documents. This is not surprising, since such documents are the traditional interface both internally across departments or projects and externally to suppliers [WW02]. In fact, the entire development process (and especially a formal contract) is based on documents and their exchange. The automotive domain has strong requirements in the development process. System specification and system development have to be consistent. , in this thesis, we argue that the use of requirements documents as input to the EvoSPL approach guarantees two main outcomes: (1) generalizes the use of our approach, since such document is of central importance when developing software in the automotive domain, and (2) reflects the automotive domain consistently, which helps to systematically manage variability among a products family of this domain.

Given the importance of requirements documents as an initial development artifact, we argue that extracting variability from such an artifact helps to map features to the other artifacts (i.e., the reference architecture and code). Since our research work focuses on the automotive domain, the Controller Area Network (CAN) Matrix document is also an acceptable input format. More explanation will be offered for the CAN Matrix in the cases study chapter (chapter 8).

There are different kinds of requirement artifacts in terms of text and various kinds of models, like textual requirements specification, user manuals, use cases, and FMs. Traditional requirements models (e.g., use cases) in their basic forms are mostly not able to document variability as required by SPLE [PBDL05]. FMs are one of the most abstract

popular ways to model variability in SPLs. In domain analysis, FMs can be used by requirements analysts to negotiate the capabilities of the system with the users [Kan+90]. Therefore, requirements documents are the most important source of information that may be used as an input to the EvoSPL approach to derive an FM.

For the rest of this thesis, we refer to the textual requirements documentation and NL that are used as primary inputs to the EvoSPL approach as requirements documents (e.g., SRS, RSD, or CAN Matrix).

Variability in textual Requirements Textual requirements express variability by certain keywords or phrases. Example 4.1 illustrates text fragments 'either' and 'or' in which requirements variability has been made explicit by highlighting the 'connectivity' variation and its variants 'Bluetooth' or 'WI-FI' methods.

Example 4.1: variability in textual requirements
The 'connectivity' of mobile phone shall be equipped with either 'Bluetooth' or 'WI-FI' methods.

4.6 Approach process overview

The process diagram in Fig. 4.4 shows how to handle evolution of an SPLs in the EvoSPL approach. The process of our approach includes activities (the main activities have a white background color, and pre- and mid- processing activities have a gray background color) taking care of migration, adaptation, and change. The process begins with the initial (prepossessing) activity study domain according to a particular products family, like the domain-specific issues (i.e., notations, techniques, or process steps) [WW02]. Normally, in the automotive domain, there are many requirements documents used to specify sensor and customer needs. Hence, we take into account the selection of the proper artifacts of the intended family, which results in (another pre-processing) activity elicit artifacts.

The difference analysis activity captures and identifies the similarities and differences between two products of a family. This activity (i) writes the requirements document (e.g., SRS) of each product into a set of atomic requirements (ARs), (ii) gives each AR a unique id, and (iii) stores ARs of each product in the so-called requirements specification (RS). Furthermore, (iv) this activity uses a proper text-based comparison tool to perform

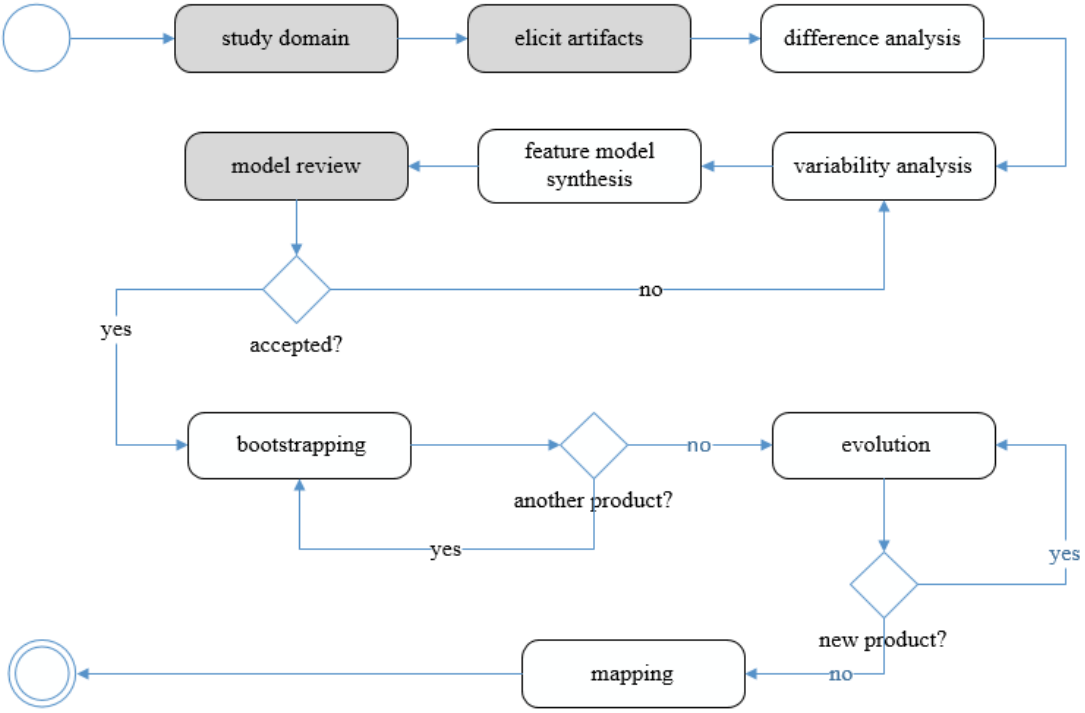


Figure 4.4: EvoSPL approach main activities.

pairwise comparison to specify the common and optional ARs between the RS of each product. The comparison concerns the changes between them in terms of matched, added, deleted, and modified ARs.

The variability analysis activity performs further processing and identifies the commonality and variability between the two products (i.e., using the RS of each product). The variability analysis activity (i) specifies common and optional ARs, (ii) stores them in a variability document (VD) of each product, and finally (ii) stores them in a SMD. A SMD represents an initial adequate view of the commonality and variability of a products family, since the products are related to the same family and share the same software architecture (reference architecture). Furthermore, this activity (iii) transfers and organizes the variability information into a FL. The feature model synthesis activity maps the variability information from the FL into the current FM, and it derives and synthesizes the initial version of an FM, which is called at this point the current FM. Once this activity has been completed, the model review activity is performed by software engineers, to confirm that the current FM models the initial SPL.

The bootstrapping activity aims to add the remaining products of a family (one by one) to the initial SPL ending with the resulting SPL. This activity (i) uses the requirements

document of a given product, (ii) derives the features of that product, (iii) stores them in a FL, and finally, this activity (iv) refines the current FM to encompass features of the product. The bootstrapping activity uses the feature model refactoring scenario, which applies predefined steps to refine the current FM with the features of a (new) product. The scenario uses a catalogue of sound FM refactorings to perform the transformations that improve and increase configurability of the current FM [Alv+06b].

The evolution activity evolves the bootstrapped SPL to encompass a new product once it has been bootstrapped. This activity aims to add the features of a new product to the current FM. It (i) uses the requirements document of a new product, (ii) derives features of the product, (iii) stores them in a FL, and finally, this activity (iv) evolves the current FM to encompass the features of a new product using the feature model refactoring scenario. Continually, this activity updates the current FM with features of a new product upon request. The activities of bootstrapping and evolution require the feature model refactoring scenario. These activities aim to create the resulting SPL and evolve it. As a side result, they support change and evolution. Finally, the mapping activity relates a feature with its code-level implementation, through a kind of tracing links. This activity uses the reference architecture of a products family as an intermediate artifact to establish the links. Using this activity, the tracing links are stored in the traceability tree, and are updated whenever feature changes occur.

Furthermore, Fig. 4.5 depicts the process diagram that relates the EvoSPL approach main activities to (1) the phases and (2) the main artifacts used, created or modified by each activity. The EvoSPL approach assumes the existence of a products family and their artifacts including the requirements document, reference architecture, and code. The artifacts and the other documents, like a SMD, an RS, and the current FM go through different activities. Different phases in this process are reverse engineering, forward engineering, and mapping. These phases are represented on the diagram by partitions rendered as horizontal 'swimlane diagram'.

4.7 Software Product Line factors and the EvoSPL approach

There are three main factors at play in SPL engineering: 1) commonality and variability management that aims to identify commonality and variability of requirements; 2) traceability of commonality and variability from requirements to the code; and 3) managing

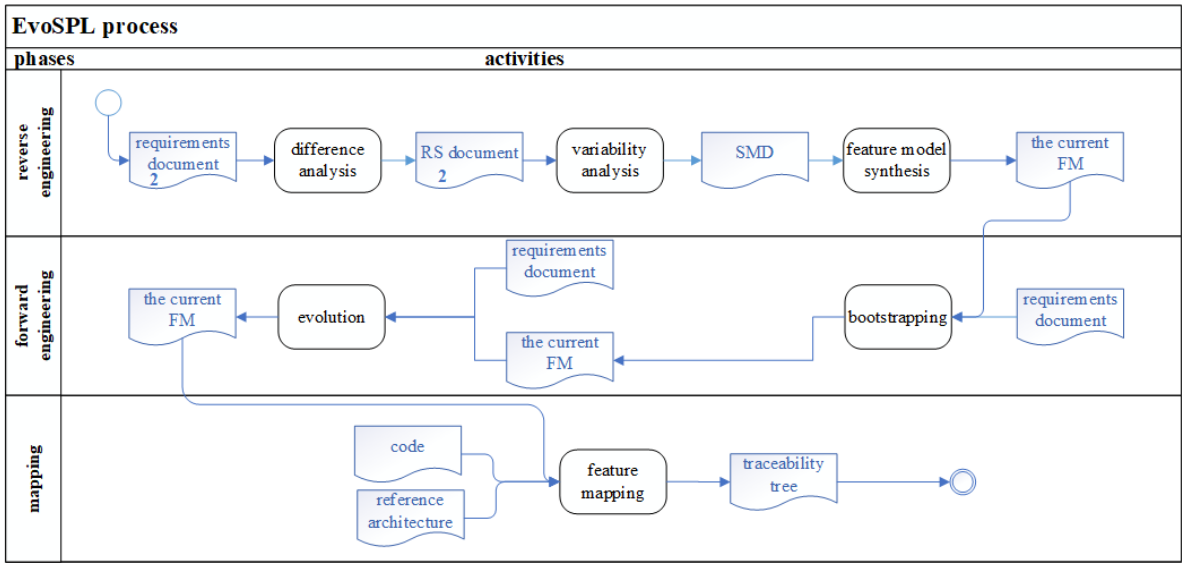


Figure 4.5: EvoSPL process diagram relates the main activities with the phases and the main artifacts.

and tracking reuse of code among the products of a family, usually driven by the previous two factors. The approach presented in this thesis proposes a re-engineering approach that consists of three main phases: 1) reverse engineering; 2) forwarded engineering; and 3) mapping. The phases of the EvoSPL approach contribute to an SPL in the automotive industry that takes the factors mentioned in this section into consideration.

This thesis deals with variability at the requirements-level of a products family in the automotive domain. Besides, it considers the code-level when applying the mapping phase, in specific, it focuses on the code implemented by the industrial C-C++ programming language, because they are the most widely used languages in the automotive domain. Variations are implicitly modeled by implementing C-C++ pre-processing directives. In this way, variable (conditional) compilation results in a specific product variant. Support for variations at code-level are provided by a few numbers of concepts and tools [Men+09], but they do not consider the specific requirements of the automotive domain. The EvoSPL approach considers such documents.

4.8 The site map of the approach

The EvoSPL approach helps software developers in handling variability and evolution of a products family. For this purpose, the approach supports a three-phase process. As shown in Fig. 4.6, the first and second phases (reverse engineering and forward engineering) cover

the problem space. The final phase (mapping) covers the solution space. The problem space includes requirements documents and the current FM artifacts, and it helps to capture and manage the variability of a family. Furthermore, it supports the possibility to deliver a specific product.

The mapping phase links the problem space to the solution space, including the reference architecture and code. The mapping phase is supported by a tool that aims to relate artifacts of the problem space to artifacts of the solution space. Furthermore, the mapping process aims to create the relationships between the artifacts of the problem space with the ones of the solution space and keep them updated. This is a concern that is continued in two different scenarios (1) when linking the artifacts for the first time and (2) whenever changes occur in the problem space artifacts.

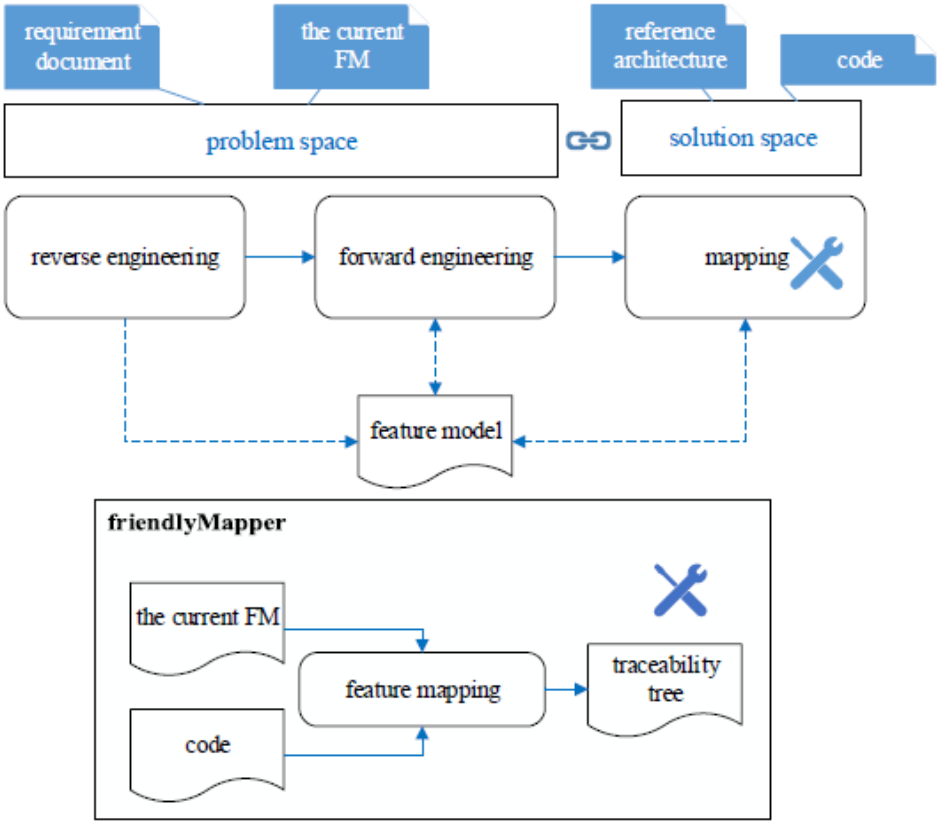


Figure 4.6: The site map of the EvoSPL approach.

Our approach aims to shift the work steps to the FM, which helps to manage variability at a high-level of abstraction and to avoid working on the code that gets more difficult to understand, to maintain, and to integrate changes. For instance, during the investigation

of the artifacts of a products family at Bosch Car Multimedia company, while performing the case study, we have found out that identifying the variability that is scattered in the code is very hard and time consuming.

4.9 Tool support for feature mapping

As Fig. 4.6 illustrates, among the three phases of the EvoSPL approach, only the mapping phase is supported by a tool developed specifically to automate the mapping activity of this phase. The tool, named friendlyMapper, semi-automatically maps the current FM to the code artifacts that are related to the products family under consideration. The tool focuses on evolution of the current FM of a products family, its implementation code, and the mapping between them using the reference architecture as an intermediate artifact.

To preserve the consistency of a given FM, the friendlyMapper accepts several operations, like adding a feature and removing a feature that the software engineers perform when they wish to change that FM. The friendlyMapper can read the current FM (i.e., XML file). In general, the changes to the current FM can have impact only on itself (e.g., changing an optional feature to mandatory or mandatory to optional), or they can also affect the mapping links (e.g., add feature or remove feature). In the latter case, the tool automatically updates the mapping to keep them consistent.

4.10 Assumptions and limitations

This section presents the assumptions and the limitations of this thesis. First sub-section clarifies the assumptions that are required as a precondition to conduct this thesis (see Fig. 4.7). Then, the next sub-sections discuss the limitations of the approach and the automated tool, which only covers the third phase of the approach (see Fig. 4.8).

4.10.1 Assumptions

Feature The EvoSPL approach assumes that the feature represents a functional unit that acceptable to users and developers, satisfies the requirements, and represent the characteristics, actions, functions, technologies or services of a products family [XXJ12] [ESSD14] [IRB14].

Our approach assumes features represent decisions and provide configuration options (the products of a family). When generating a products family, the features can be common (shared by all the products of a family) and optional (differ between the products of a family) [Dav+13]. In consequence, a set of products generated from a systematic modeling of features is called an SPL.

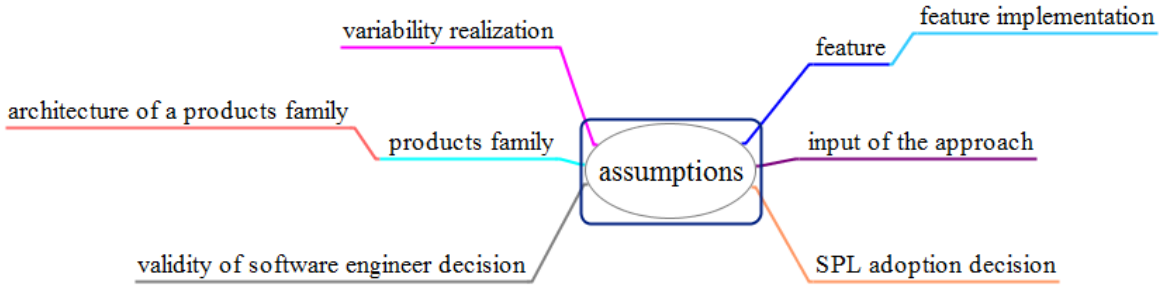


Figure 4.7: The assumptions of the EvoSPL approach.

Feature implementation The approach considers that a feature is implemented at the code-level, using the industrial C-C++ programming language. Thus, features are implemented based on the project hierarchical level of the code elements (according to programming grammar), ranging from the higher hierarchical implementation level, such as (sub) packages, source files, and routines, to lower hierarchical implementation levels, such as statements and expressions.

The EvoSPL approach assumes that a feature is implemented with a set of code fragments (feature-related-code fragments) of the code elements mentioned above. Besides, a feature always has the same implementation in all products where it is present. The approach also assumes that a feature-related-code fragment includes an implementation associated with more than one feature.

A products family The EvoSPL approach assumes that the products of a family and the new one, in the automotive domain, are created with ad-hoc reuse (i.e., cloning or branching). It assumes also that the implementation of the products of a family at the code-level is written in C-C++ based language. It is also assumed that the domain requirements are described using NL (e.g., 'English') and presented in a textual format, such as SRS documents [BKS15].

Using our approach, it is assumed that a set of existing products family is often available as a starting point for building the resulting SPL. Additionally, the understanding of how features evolved in the products family is a prerequisite of the transition from ad-hoc into the systematic reuse using an SPLE. Our approach assumes the process of cloning (or branching) a (new) product of a family is performed using the initial release product. The initial release product is a product that often evolved from the platform developed and successfully used by the first customer and then modified according to customer needs. However, if another product in the family is more suitable, as taken as a basis, then it can be chosen.

Furthermore, it is assumed that the minimum number of the required products of a family to perform the process of our approach successfully are three products of a family and a new one. The new product is scheduled for being developed in the near future upon receiving a new customer request.

Variability realization It is assumed that the variability of the code is handled using C-C++ pre-processor directives [Men+09], such as conditional compilation directives, macro definition, and predefined macro names and conditional execution, such as if-else and switch blocks.

The architecture of a products family The approach assumes the evolution of an SPL is performed in a setting where a family of products has a reference architecture (which represent the architecture of all products), and the products were developed without considering a systematic approach.

SPL adoption decision Our work assumes that the decision to extract an SPL from a set of existing products family has been taken by the practitioners in the automotive domain. The EvoSPL approach does not check the possibility and convenience of such a decision, like business and organizational issues. For example, the cost estimation and benefits. It is assumed all these issues are out of scope of this thesis. Our approach supports a process that helps to adapt an SPL in the automotive domain, which may influence practitioners to take the decision to adopt a systematic reuse using SPLE and to avoid the precautions of such adoption.

Validity of software engineer decision Our work assumes that the software engineer can make valid decisions when required by the process of the EvoSPL approach. For instance,

concerning the name of the identified feature, software engineers can manually confirm or assign a name of the identified feature. The approach does not support any validation of the software engineer decisions (e.g., when they approved the output models).

Input of the approach It is assumed that each product of a family has a known requirement document presented in the NL and used as a main input to our approach [IRBW15]. Our approach assumes that the requirement document may occur in various forms. According to review in [LSS17], our approach assumes that SRS/RSD will be frequently used as the primary input artifact to extract variability and identify features. Besides, it also assumes that some additional information in a tabular format is accepted.

4.10.2 Limitation of the approach

Automation The adoption of SPLs can be fully automated, completely manual, or somewhere in between. The automation of the EvoSPL approach, in this thesis, is limited to adoption of the tools that are already developed/available and used by another research (e.g., [Thü+14] [Difa]). Exceptionally, the third phase of the approach is partially automated. Due to the fact that the automotive domain has special requirements, the software engineers in this domain have frequently been dealing with variability in documents in the past, and they know which manual adaptations have been more relevant. Thus, for two reasons, some steps of the EvoSPL approach are performed manually. The first reason, technically a fully automated SPL adoption is not possible in general. The second one, to allow the intervention of software engineers in the process of EvoSPL approach when needed or required.

The derived FM In our work, the derived FM does not define all the features of a product family and all the constraints among the features with 100%. The reason behind this limitation is that some features and their selection constraints are not detected. Our work limits the explanation of dependency-relationships among features of the current FM to mentioning it in through the manuscript. It does not present this relationship in the illustrative example or industrial case study.

Generality of the approach The evaluation of the EvoSPL approach is conducted only in the automotive domain, we argue that the process and method addressed here are valid for the automotive industry in general. We also conducted one case study in the

automotive domain, but we believe that other highly variant domains could benefit from our approach.

Difference analysis technique The EvoSPL approach can handle pairwise comparison of a products family. The approach compares and analyses requirements documents of two products at the same time. However, the approach adopts a tool that can compare two (up to three) documents together. Our approach enables software engineers to use any tool that compares two versions of a document and specifiers the parts that have remained the same and the difference parts between them (e.g., [Difa], [Dra]).

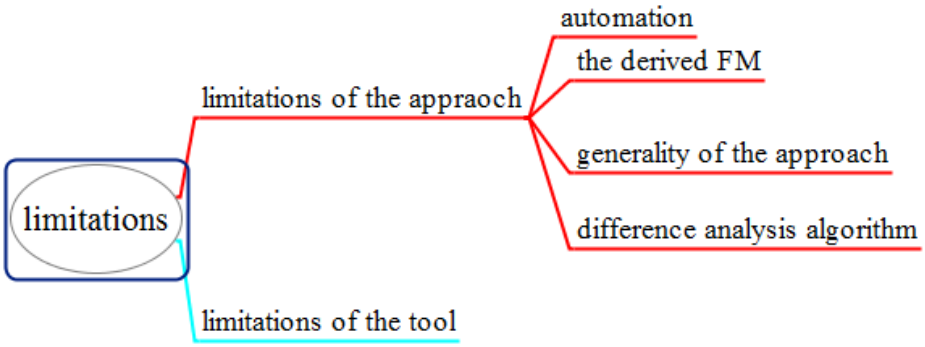


Figure 4.8: The limitations of the EvoSPL approach.

4.10.3 Limitation of the tool

The friendlyMapper tool presented in this thesis uses a particular artifact (i.e., traceability tree) to map explicitly every feature of the current FM to the code fragments that implement this feature (at the code-level). The tool is not able to map a feature to its places in the reference architecture. The tool can automatically read features of the current FM from XML file and maintain them in the traceability tree. At the same time, the tool is not able to access the code fragments that implement a feature. Thus, it requires the software engineer to manually enter the name of the feature-related-code fragments (i.e., routine name) in the traceability tree.

The tool uses tracing links that relate and store the feature to its implementation code fragments in the traceability tree and updates the tracing links whenever feature changes occur. However, the automatic update of the code is required to maintain its coherence with features. This is a limitation of our approach that we plan to address in the future.

4.11 An illustrative example: ATM products family

As an illustrative example, we consider four products of the ATM family. Three products of a family and a new one. The new product is scheduled for being developed in the near future upon receiving a new customer request. These software products are developed by ourselves. It is worth mentioning that these software products are developed by clone-and-own technique based on the initial release product [San]. The purpose of this example is to illustrate the main phases of our approach (which are explained in the upcoming chapters). The ATM family allows a user to perform different kinds of ATM transactions. The products of an ATM family represent a small case study.

As shown in Table 4.1, Product 1 supports common features among all the products (core features): 'check_balance', 'withdraw_cash', 'deposit_cash', and 'quit_transaction'. Product 2 supports 'convert_currency' together with the core ones. Product 3 has the core features of Product 1 and new 'show_balance_screen', 'withdrawal_cash_limit' and 'transfer_money' features. ProductNew supports 'show_balance_report' and 'pay_service' features (in addition to previous optional features except 'convert_currency' the feature), together with the core ones.

The FM in Fig. 4.9 shows the current FM of the ATM products family as manually designed using FeatureIDE editor. This FM represents a basic FM without cross-tree constraints. In this example, we used the products of ATM family (and the related artifacts that we have developed) to better explain some parts of our work (we use the same example to explain the steps of the EvoSPL approach in the upcoming chapters). However, we only use the requirements documents, reference architecture, and code of the products of the ATM family as an external input of the EvoSPL approach and thus we do not know features in advance.

Table 4.1: Features set of each product of the ATM family.

product name	features of the product
Product 1	{check_balance, withdraw_cash, deposit_cash, quit_transaction}
Product 2	{check_balance, withdraw_cash, deposit_cash, quit_transaction, convert _currency}
Product 3	{check_balance, withdraw_cash, deposit_cash, quit_transaction, show_balance_screen, withdraw_ cash _limit, transfer_money}
ProductNew	{check_balance, withdraw_cash, deposit_cash, quit_transaction, show_balance_screen, withdraw_cash_limit, transfer_money, show_balance_report, pay_service}

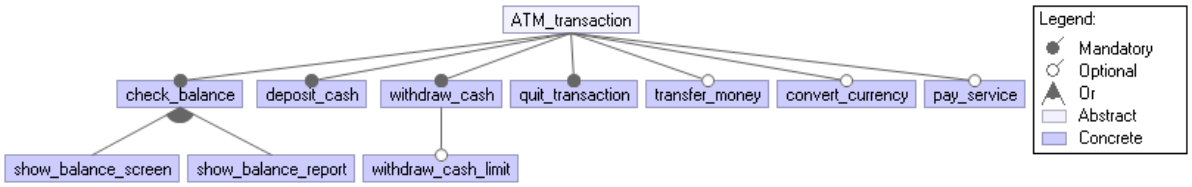


Figure 4.9: The current FM of the ATM products family.

Chapter 5

EvoSPL: Reverse engineering phase

The adoption of an SPL requires (1) analyzing commonality and variability of the existing artifacts and (2) tracking the changes that were introduced to these artifacts. This chapter describes the reverse engineering phase of the EvoSPL approach that contributes to support what is required in (1) when adopting an SPL practice. The reverse engineering phase is divided into three main activities that aim to identify and document the commonality and variability of the products of a family in the form of a variability model (i.e., the current FM). This phase allows for managing the variability of the products of a family at requirements-level using requirements documents artifact. The main output of the reverse engineering phase is the current FM, which has been derived using the activities of this phase.

As shown in Fig. 5.1, the EvoSPL approach starts off with the reverse engineering phase, which uses the difference analysis activity and variability analysis activity to specify what the common is and variable between two products. Besides, it uses the feature model synthesis activity to deliver the current FM for the initial SPL. This phase is conducted through a variety of semi-automatic approaches to define the common and variable requirements of a products family and to derive and synthesize an FM, namely, the current FM of an SPL under consideration. Furthermore, this phase introduces three novel methods. The first one is a feature identification method, the second one is a feature naming method, and the last one is dedicated to an FM construction.

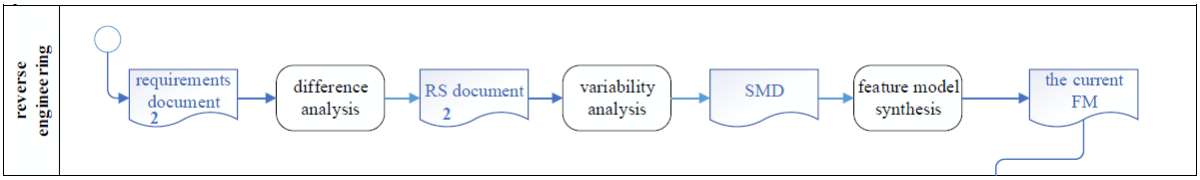


Figure 5.1: EvoSPL process: reverse engineering phase.

Modelling variability in the EvoSPL Modelling variability is an important step in an SPL practice. There are many different approaches of variability modelling, each with a different focus and goal. The EvoSPL approach adopts the common approach that aims to express variability in terms of common and optional features, a process called feature modelling. It uses FMs (called in this research the current FM) and their graphical representation as feature diagrams, because they are currently the most popular form of variability models.

The EvoSPL approach uses the reverse engineering phase to derives the current FM that owns the following characteristics.

- In characteristic 1, the current FM can be used by software engineers to negotiate the capabilities of the product with the users (in case the requirements documents are within the domain of the products). Our approach focuses on extracting the functional features that represent the customer needs.
- In characteristic 2, the current FM identifies Common, Optional, Or, and Alternative features.
- In characteristic 3, the current FM identifies the variability-pattern (i.e., Common, Optional, Or, and Alternative) of each feature.
- In characteristic 4, in addition to the variability-pattern, the current FM identifies feature-relationship. The parent-child relationship between features and dependency-relationship (cross-tree constraints) are allowed.

All the characteristics are explained in detail within the manuscript of this chapter.

5.1 Difference analysis activity

This section explains the difference analysis that is designed to fit product families of the automotive domain. This activity supports the necessary analysis of similarities and differences among the products of a family using two products of a family at once. This activity consumes the requirements document of each product and delivers an initial document that represents the individual similarities and differences for each product of a family. Using a proper text-based difference analysis (e.g., a text-based comparison tool), this activity captures and detects the similar and different observed text items in the matched requirements documents.

The activity in brief Understanding the modifications from one product to another starts with identifying the differences of their specifications. As shown in Fig. 5.2, the difference analysis activity (i) takes as input the requirements documents of two products of a family, (ii) writes the requirements documents of each product into a document that adopts rules and formats, named an RS. Next, this activity (iii) detects the similarities and differences between the RS of each product, and (iv) organizes (requires software engineering intervention) the comparison results in an RS of each product.

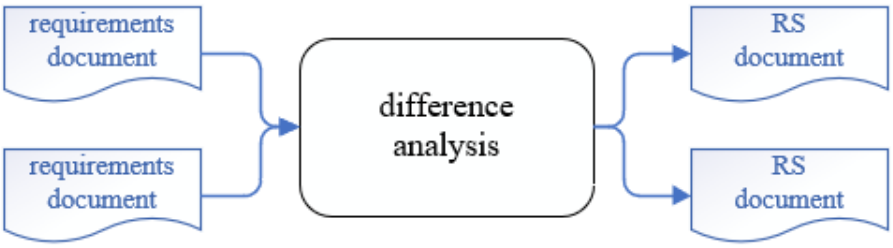


Figure 5.2: EvoSPL process: difference analysis activity.

Specific requirements of the difference analysis activity According to the EvoSPL approach, there are specific requirements to support the difference analysis activity to improve the matching between requirements documents, and they are.

R1. It is required to understand the domain of the products family and consult project manager and development team regarding the general information of the family under the consideration.

R2. Many software engineers change the names given to signals, services, and technologies while customizing the requirements document to adapt the requirements of a new customer. Changing the names indicates unrelated and irrelevant differences that can be considered as a modification for the difference analysis of the resulting SPL, when matching the requirements documents against each other. To mitigate this, it is required to unify the name of signals, services, and technologies among the requirements documents of a family under the consideration.

R3. It is required to ignore irrelevant differences for the variability of the resulting SPL, like filtering formatting changes and editing beautifying of requirements documents.

R4. It is required to specify the relevant variability items to be considered when matching requirements documents. This can be determined by software engineers during the study

Table 5.1: An example of relevant variability items of the requirements document in the automotive industry.

variability item	examples
document release information	date, product number, revision number
message structure	message layout, number of bits, error handling
message signals	signal label, signal description, signal designation, signal status, signal counter, signal values

domain activity. Table 5.1 lists an example of relevant variability items in requirements documents of the automotive domain that are considered by our approach. We do not claim for completeness, but we believe they are valuable in the matching process.

R5. It is required to guarantee that the products of a family used by our approach are reliable and working, compiled, and executed independently.

R6. Before starting the difference analysis activity of the EvoSPL approach for the first time, it is required that software engineers select one of the products to match as the initial release product and another one needs to be the product with the most functionalities (features) among all the products of a family. The software engineers of a family can decide about the products in the domain analysis and artifacts election activities.

The difference analysis steps Using two products of a family, this activity consumes the requirement document of each product to apply the difference analysis steps that are defined for the EvoSPL approach as follows.

Step 1.1 rewrites the requirements document of each product into a set of ARs structure.

Atomic requirement structure AR in this research work is an object with a unique identifier (id) and self-contained statement (statement) about the properties and functionalities of a product under consideration. To rephrase the text of requirements document into AR structure, our approach adopts a specific format (guidelines) that is presented by Fernandes et al. in [FM16]. The following examples show RS that follow the guidelines.

atomic requirements	
id	statement
R1	Perform ATM transactions.
R1.1	ATM checks balance.
R1.1.1	ATM shows the balance on screen.
R1.1.2	ATM shows the balance on report.
R1.2	ATM withdrawals cash.
R1.2.1	ATM withdrawals cash with a limit.
R1.2.2	ATM withdrawals cash without limit
R1.3	ATM deposits cash.

Step 1.2 stores the new structure of the requirement document into an RS document. The RS is then considered for comparison with the RS of another product.

RS preparation Using the EvoSPL approach, software engineers can manually edit and review the ARs of an RS using a proper word processor (e.g., document formats of a popular office suite like Microsoft Word can be used). The software engineer experts who are familiar with the varying among a products family need to analyze such existing documents. Such experts have frequently been dealing with variability in documents in the past and they know which parts in the text have most relevant details that may contain possible variations [Rab+08].

Step 1.3 performs a text-based (line-based) comparison that matches the RS of each product against each other, to specify the similar and variable ARs between them inform of change terms. The result is the similar and different ARs between the matched RS documents (distance between the documents). By the end of this step and for simplicity, the matching results are stored in the RS document of each product. Since the similarity and variability are specified using change terms beside each AR, this reduces redundancy and avoid inconsistencies between the documents.

Change terms of the ARs According to EvoSPL approach, RS of each product are scanned and compared to specify the changes between them. The comparison concerns the change in terms of matched, added, deleted and changed ARs, which called change terms of the ARs. There are no restrictions regarding the technique used for text-based comparison. However, the technique should be flexible to highlight both the similarities and the differences between the documents (e.g., RS documents).

According to the EvoSPL approach, matched ARs identify similar requirements existed in both RS documents (both products). The three types of differences added, deleted and changed ARs

identify the modifications between two products. Both added and deleted identify differing ARs existed in one of the RS documents, where changed identifies differing ARs existed in both. The concept of change terms of the ARs is used to determine the variability-pattern of the AR (see Variability-pattern of the AR of section 5.2.1)

RS structure Fig. 5.3 depicts the structure of an RS by the end of the difference analysis activity. Each product has one RS that contains many interdependent ARs, where each AR has a unique id and written statement. AR is the building blocks of each RS and it presents the similarity and difference (the change terms) of the products that have been selected as an input to the difference analysis activity. Moreover, a set of ARs may specify a feature and the relationships among them, which correspond directly to the notations used to build the current FM in future and which enable a clear understanding of the features of a products family.

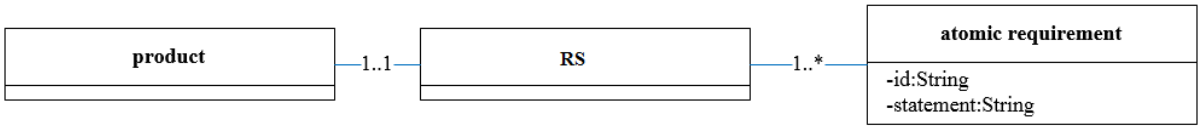


Figure 5.3: EvoSPL process: requirements specification structure.

5.2 Variability analysis activity

This section describes the variability analysis activity dedicated to the EvoSPL process, to support the variability management of an SPL. As shown in Fig. 5.1, the variability analysis activity follows the difference analysis activity. Typically, it is the basic activity when adopting an SPL from the existing products [PBDL05]. The main purpose of the variability analysis activity is to support the design of variability of the products of a family iteratively into the current FM, by identifying the commonality and variability among them. The variability analysis activity aims to create a FL that describes the commonality and variability of a products family in terms of features and decides about its representation in the resulting SPL in a given context. The outcome of the variability analysis activity is used to feed the feature synthesis activity. This activity derives and constructs the current FM owned specific characteristics explained earlier in this chapter (see Modelling variability in the EvoSPL).

The activity in brief In general, the variability information at requirements-level is organized in the requirements document artifacts of a products family. In the variability analysis activity, important observations can be made to specify difference analysis between multiple products in the context of variability. As shown in Fig. 5.4, the variability analysis activity (i) receives the RS documents of the current used products (the products of a family that were used in the difference analysis activity) as an input, (ii) specifies the commonality and variability between them, as

well as stores the result in a VD of each product, and finally (iii) returns the commonality and variability of the VDs into a SMD (see VD and SMD). This leads later to (iv) identify features and store them in a FL. This activity is the longest in the reverse engineering phase and it consists of four main macro steps: variability presentation, visualization technique, features identification, and variability transformation.

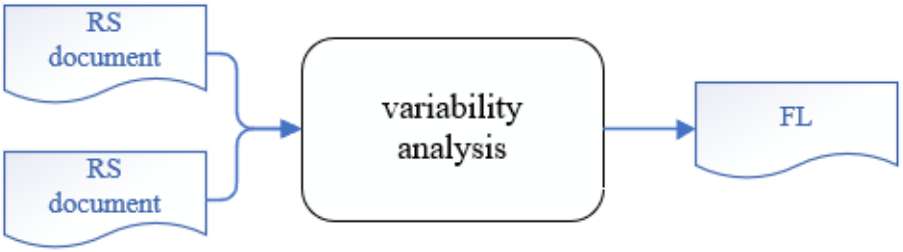


Figure 5.4: EvoSPL process: variability analysis activity.

VD and SMD Equivalently, a VD and SMD contain the analysis of commonality and variability of the products of a family at requirements-level. Compared to the SMD, the VD involves the individual information related to commonality and variability of a specific product of a family (commonality and variability on product-level), where a SMD represents an initial adequate view of commonality and variability of the products family (commonality and variability on a products family-level).

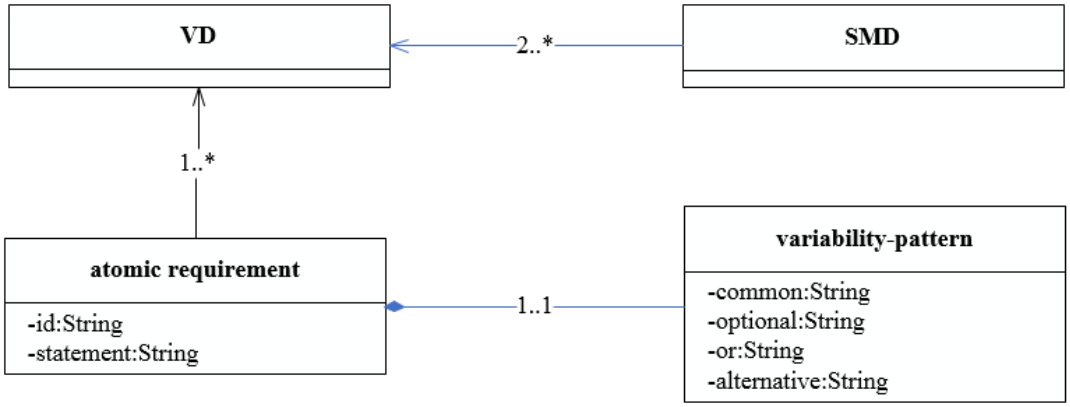


Figure 5.5: EvoSPL process: A variability document and single master document structure.

5.2.1 Variability presentation macro step

The variability presentation macro step reads from the VDs of two products and uses the visualization technique (section 5.2.2), which proposed for the context of the EvoSPL approach, to

organize commonality and variability into a SMD. This is based on the facts that the products of a family are related to each other and have enough similarities. In addition, they almost have a reference architecture. Fig. 5.5 depicts the structure of a VD, SMD, and their interrelation by the end of variability presentation macro step.

When the matching in the difference analysis activity is done, the resulting matched documents (the RS documents) are further processed in the variability analysis activity to identify the actual commonality and variability of the products family (see Table 5.2). The latter activity adopts the variability presentation macro step to guide a consistent variability design, by comparing the VD of each product against each other, identifying the common and variable parts (ARs) between them, and then storing the results into a SMD. The following are the variability presentation steps.

Step 2.1 assigns the variability-pattern of each AR and stores them in a VD of each product.

Variability-pattern of the AR The 'common' ARs represent the commonality (common textual requirements) among the products of a family and the 'optional' ones represent the variability (variable textual requirements). Based on the definition of change terms of the ARs: matched, added, deleted, and changed presented earlier in this chapter (see Change terms of the ARs), the EvoSPL approach defines the variability-pattern of the AR as explained in Table 5.2.

Step 2.2 compares (matches) the VD of each product against each other.

Step 2.3 transfers the 'common' AR into a SMD without alteration and labels the AR with a 'common' keyword.

Step 2.4 transfers the 'optional' AR into a SMD and labels it with an 'optional' keyword. Furthermore, it specifies the VD and the product that the AR belongs to.

Step 2.5 transfers the 'or-group' / 'xor-group' AR into a SMD under its parent AR (see Parent AR below) and labels the AR with 'or-group' / 'xor-group' keyword. Furthermore, it specifies the VD and the product that the AR belongs to.

Step 2.6 repeats step 2.3 to step 2.5 until the final AR is reached.

Parent AR In the EvoSPL, a parent AR is an AR that contains all the ARs contributing to the same textual entity. The entity represents a heading (title), type, category, technology-specific, signal, service that appears in the requirements document of a product and transfer to

Table 5.2: Variability-pattern of the atomic requirement.

variability-pattern	change term	description
'common'	matched	AR is included in both RS documents.
'optional'	added, deleted, changed	AR is part of a group of ARs that are existing in the other RS documents, and if their parent's AR is included, at least one of those ARs is included in the RS.
'or-group'	changed	AR is part of a group of ARs that are existing in the other RS documents, and if their parent's AR is included, at least one of those ARs is included in the RS.
'xor-group'	changed	The AR is part of a group of ARs that are existing in the other RS documents, and if their parent's AR is included, exactly one of those ARs is included in the RS.

the VD as a side result of the difference analysis activity. Typically, the ARs have shared terms among each other and with the parent AR that they belong to. The software engineers can use the text-based parsing (see Text-based parsing below) to investigate such terms. The parent AR can be mandatory, if it includes requirements from all input VDs or optional otherwise. The EvoSPL approach uses a text-based parsing to extract terms from the ARs separately and identify the relationships among them, if they share similar terms. These relationships are then used to investigate the relationships among ARs and to recommend a grouping of the related ARs.

Related ARs The visualization technique and the feature identification macro steps require identifying the related ARs and parent AR. Thus, the EvoSPL approach supports investigating the relationships among the ARs using the following resources: (1) domain analysis of the related products of a family, which performed during domain analysis activity, (2) requirements documents, which is originally used as an input to the difference analysis activity, (3) the domain requirements engineers can often help to identify the related ARs, and (4) the text-based parsing. The EvoSPL approach requires the software engineers to provide a final decision to confirm about grouping of the ARs.

Relationships among the ARs In the EvoSPL, software engineers can investigate the relationships among the ARs to group them together and then to identify the parent AR. Listening 1 of the illustrative example (section 5.4) explains this concept for two ARs. In particular, it is recommended to analyze the relationships among the ARs of a SMD (all the explanation is also applicable for the documents that adopts the AR structure) iteratively, where a single iteration comprises of (i) scanning the ARs of a SMD, (ii) performing initial grouping of the ARs, (iii)

investigating the relationships among the ARs to identifying the related ARs and the parent ARs.

Initial grouping of the ARs The software engineer can initially apply a grouping to any set of ARs of a document that adopts an AR structure. For example, grouping two or more ARs that have shared terms or dependency. The key of the following recommendations is to enable software engineers in their role to group ARs

1. Software engineers can group the ARs originated from the same section or subsection of requirements document into a group.
2. Software engineers can group the ARs originated from the same paragraph of requirements document into a group.
3. Software engineers can group the ARs originated from a consecutive order of sentences of requirements document into a group.
4. Software engineers can group the ARs originated from a compound sentence of requirements document into a group. Normally such a sentence represents or/alternative ARs that are related to a parent AR.
5. Software engineers can group the ARs originated from sentences that explain a title (or heading) of requirements document into a group.
6. Software engineers can group the ARs that originated from sentences explaining a service, technology, signal, action or scientific term of the requirements document into a group.
7. Software engineers can group the ARs that have shared terms and dependencies into a group.

Investigating the relationships among the ARs Considering the results of an initial grouping of the ARs (see Initial grouping of the ARs), software engineers can investigate the relationships among ARs in the same group, to find out the related ARs (and parent AR), as follows.

1. Software engineers scan the ARs in the same group separately.
2. Software engineers use the text-based parsing (see Text-based parsing) to extract terms from each AR (see Extract terms from the ARs).
3. Software engineers build a set of terms of each AR and try to find commonalities (e.g., shared terms) among them.

4. Software engineers study and specify the relationships among the terms of each group of the ARs.
5. Software engineers identify related ARs and then determine the parent ARs.

Text-based parsing Our approach uses a text-based parsing technique to extract terms from the ARs separately (i.e., NLP). First, the software engineers identify the sequences of terms (important words that have variability – related information, present functionality, or feature) in each AR of the same group. Each term forms a unit representing a token defined in the grammar of a sentence. Thus, the parsing provides a set of terms of each AR represented in a document (e.g., a VD or SMD). The experts who are familiar with the variability among the products of a family need to analyze the terms. Such experts have frequently been dealing with variability in the documents in the past and they know which terms have most relevant details that may contain possible grouping of the ARs. At this point, the experts can extract terms from each AR in the same group and detect the relationships among them.

Extract terms from the ARs Software engineers can 1) parse (i) functions (ii) actions (or verbs), (iii) objects, including instruments, technologies, services, and signals, (iv) parameters (or attributes), but they can ignore their values unless the value influences an action (e.g., a value that influence calibration of the sensor), and (vi) capabilities. Also, characteristics are taken into consideration. Next, they 2) apply normalization on the parsed terms of each AR prior investigating the relationships among them.

Normalization The text-based parsing in the EvoSPL approach uses a term processing to normalize the extracted strings and receive more valuable terms [SKL08]. The proposed processing includes.

- Splitting by separating the extracted terms when required with a dash (or space) in between, like “messageSiganl” to “message” and “signal”.
- Stemming by removing the suffix from a word and reducing it to its root word, like using singular terms only “messages” to “message”, “transfers” to “transfer”, or “transferring” to “transfer”.
- Filtering by excluding the non-essential terms, like removing terms with less than three characters or stop words like ’the’, ’is’, ’at’, ’which’, and ’on’.

5.2.2 Visualization technique macro step

This section presents the visualization technique that is dedicated to the variability analysis activity of the EvoSPL approach. The technique aims to improve visibility of the commonality

and variability in the delivered documents of the approach (e.g., a VD, SMD, and FL), which consequently helps to identify features and their variability-patterns (i.e., common and optional features) in the documents [Mar+14]. The EvoSPL approach uses a semi-automated technique that requires software engineer's intervention and a word processor that is equipped with a predefined theme color palette (e.g., the theme color palette of the Microsoft Word). As shown in Fig. 5.12 and Table 5.4, the visualization technique helps to display variability information in the EvoSPL approach and supports the following capabilities.

1. The visualization technique colors the common and variable (different) ARs in green and yellow bars respectively. Each bar visualizes an AR and the length of the bar depends on the length of the statement.
2. The visualization technique uses keywords such as 'Common', 'Optional', 'Or', and 'Alternative' to define the variability-patterns of the ARs. An 'Or' and 'Alternative' symbolize 'or-group' and 'xor' group respectively.
3. The visualization technique colors the related ARs (or grouped together, see related ARs) with the same color.
4. The visualization technique colors parent-feature and its related child-feature with the same color.
5. The visualization technique assigns each feature (parent-feature and its' child-feature) the same color of the ARs that contribute to this feature.

5.2.3 Feature identification macro step

The EvoSPL approach does not assume a clear upfront knowledge of the existing features throughout artifacts of a products family. Feature identification macro step identifies features of the SMD. This macro step is applicable to any document that adopts the AR structure (See Atomic requirement structure).The feature identification macro step performs the following feature identification steps.

Step 2.7 (text-parsing) analyses each AR in a SMD and identifies its constituents with the variability term role. This step specifies the terms within each AR that have important variability information (features and dependencies among them) to the future SPL. Finally, this step results in a set of terms for each AR. Each term or a collection of terms in the set may represent a feature. This depends, finally, on domain engineer's confirmation.

Variability term role Using our approach and following some concepts presented by Ridzik and Reinhartz-berger [IRB14], software engineers can parse ARs through actions, functions, and

services. They can use the following variability terms role, which have special importance to the functional perspective of variability of the automotive industry, including: tr1. Who performs? (e.g., sensor), tr2.What is performed? (action/verb - e.g., calibrates), tr3. On what object is it performed? (e.g., angle), tr4. How is it performed? (instrument - e.g., calibration value), and tr5. What condition is performed? (e.g., flag=1). Besides, software engineers can use objects (tr3), including the following: names and types of technologies, services, and signals. Additionally, they can use tr6. capabilities, parameters, and characteristics.

Step 2.8 (naming features) suggests a proper name for the feature from the set of the extracted terms of each AR, using the naming features method (NFM).

Naming features method NFM method takes into consideration (1) the software engineers' preferences, (2) the appropriate perspectives of variability, and (3) the set of terms of each AR. Besides, this method considers specific resources when giving a proper name for the feature. The method (i) gives a meaningful feature name (see Feature name guidelines below) and (ii) consults the domain expert and project manager for the revision and confirmation. In case of rejection, the software engineers ask the domain expert and the project manager to (iii) propose a proper name and then they (iv) investigate the consistency of the name to decide on the final name of a feature. This reworked as many times as possible to (vi) resolve the conflicts concerning the feature name until the proposed name turned into an agreement.

Resources The resources that help software engineers when giving a proper name for the feature are: (i) the requirements documents of the products of a family, (ii) code, including the related modules, routines, variables or constant names, and comments – usually the code has meaningful names provided by the developers, and (iii) domain experts, domain requirements engineers, and a project manager.

Feature name guidelines Basically, it cannot be assumed to find a clear name of the feature inside the description and implementation of a products family. For the purpose of assigning meaningful names for the features that are used to build the current FM, software engineers can use the following guidelines.

1. Software engineers can use requirements documents and components of the reference architecture as a guidance to suggest a proper name for the feature. Besides, they use the code related modules, routines, variables or constant names, and comments.

2. Software engineers can follow the perspective of domain requirements engineer and project manager to suggest a proper name for the feature.

3. Software engineers apply normalization (see Normalization) on the parsed terms of each AR, prior giving a name for the feature.
4. Software engineers give a unique name for each feature.

Give a unique name The name of a feature can be any free-form text that describes the feature [XXJ10]. The software engineers can give a name of the feature as follows: irst, they can choose from the extracted terms of each AR. Next, they keep the name of technologies, hardware, instruments, signals, actions, condition flags, services and attributes of the AR without alternation. In other words, they can use them to suggest a name of the feature. In case a free-form text of the feature has two words, they can separate terms with a dash (or underscore) character in between.

Step 2.9 (variability identification) identifies and organizes the variability information of the SMD, to fit its transformation into a FL (as a pre-step to feature model synthesis activity) using the feature identification method (see Feature identification method).

Variability information The variability information of a SMD is identified and organized in three main dimensions: (i) the feature name, which is the main building block of the current FM (valid combinations of the features present a unique product), (ii) variability-pattern of the feature that identifies the type of feature and its representation in the current FM, and (iii) feature-relationship that defines valid connections between the features.

Variability-pattern of the feature The variability-pattern of the features: Common (or mandatory), Optional, as well as Or, and Alternative are determined by our approach, based on ARs of the input VDs. This can be abstracted as follows.

1. The feature can be ‘Common’, if it includes ARs from all input VDs (i.e., two VDs).
2. The feature can be ‘Optional’, if it includes ARs from one or more VDs but not all of them.
3. The feature can be an ‘Or’, if it includes AR within at least two ARs grouped under the same parent AR in a SMD and originated from one or more VDs.
4. The feature can be ‘Alternative’, if it includes AR within at least two ARs grouped under the same parent AR in a SMD and originated from different input VDs.

Feature identification method The EvoSPL approach contributes a novel method, named feature identification methods (FIM) to perform variability identification of a document that

adopts the AR structure (see Atomic requirement structure). The method identifies the features and their variability-pattern and feature-relationship in the SMD and summarizes them in a FL. Fig. 5.6 depicts the main stages in the FIM: foundation, variability knowledge, variability rules, and process.

Stage 1 (foundation) is a collection of basics that constructs FIM and facilitates its applicability on the ARs of a SMD or document that adopts the AR structure.

F1 FIM is built on domain analysis of the existing product families.

F2 FIM uses an FM concept in the SPL.

F3 FIM is somewhat based on the inference rule methods, specifically, the forward chaining [Chi06] [RJN16].

Stage 2 (variability knowledge - VK) is a collection of facts that are applied sequentially to each AR of a SMD (see below).

1. A set of ARs identifies a feature.
2. A parent AR forms a parent-feature.
3. A set of ARs forms a 'Common' or an 'Optional' solitary feature, whenever they have one value and do not carry options or alternatives.
4. A set of common ARs represents a 'Common' feature.
5. A set of variable ARs represents an 'Optional' feature.
6. Grouped ARs except the parent AR in the same group form child-feature.
7. AR forms a 'feature group' feature whenever it consists of a set of 'or-group' or 'xor-group' ARs.
8. AR forms an 'Or' feature, whenever it is within a group of ARs ('or-group'), and at least one or more ARs (options) can be included in a product. This can be observed by software engineers while scanning the SMD.
9. AR forms an 'Alternative' feature, whenever it is within a group of ARs ('xor-group'), and just one AR (option) can be included in a product. This can be observed by software engineers while scanning the SMD.

Stage 3 (variability rules - VR) are a collection of rules that have an if-then statement format. The if-then statement consists of two sides, on the left-hand side is if-side and on the right-hand-side is then-side. We can apply the rule on an AR whenever the left-hand side of a VR matches a given AR (see below).

R1: If an AR is a parent-feature, then it has a parent-child relationship with features of the same group.

R2: If an AR is a child-feature, then it has a parent-child relationship with the parent-feature of the same group.

R3: If an AR has an 'or-group' variability-pattern, then it forms an 'Or' feature with the 'feature group' of the same group, and it is contained by an 'or-group' with the ARs of the same group.

R4: If an AR has an 'xor-group' variability-pattern, then it forms an 'Alternative' feature with the 'feature group' of the same group, and it is contained by an 'xor-group' with the ARs of the same group.

R5: If an AR requires another AR to be included, then it has a 'requires' relationship with that AR.

R5: If an AR requires another AR to be excluded, then it has an 'excludes' relationship with that AR.

Stage 4 (process) works on the ARs of a SMD and updates them sequentially. As shown in Fig. 5.6, FIM starts the first iteration by using the VK and applies them sequentially on each AR until the last AR in the SMD is reached. In case, one of the VK is not satisfied (does not match an AR) it will be skipped. After the ARs of the SMD are updated by the VK, the FIM starts the second iteration, by using the VR. The method searches ARs until one of them matches the left-hand side of one VR and then applies the right-hand-side of this VR on that AR. The FIM stops whenever it reaches the last AR in the SMD. Now the SMD is updated with variability information that makes it rich enough to feed the variability transformation macro step.

5.2.4 Variability transformation macro step

The variability transformation macro step transfers the identified variability of a SMD (or a document that adopts the AR structure) and organizes it into a FL in terms of features. FL is

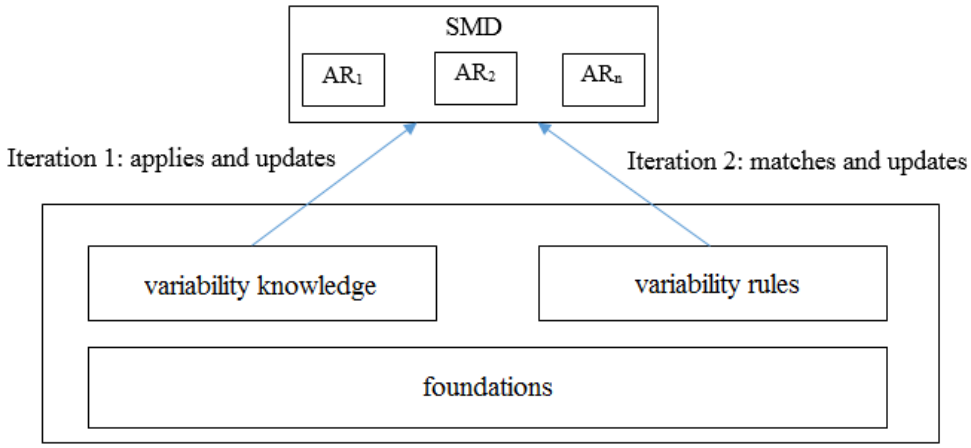


Figure 5.6: EvoSPL process: process structure of feature identification method.

a list of features with the references to the ARs that have/implement each feature. Besides, the list presents the variability of each feature in a three-dimensional format (1. feature name, 2. variability-pattern, and 3. feature-relationship). Thus, the content of a FL is used directly as an input to feature model synthesis activity. This macro step adopts the following variability transformation steps.

Step 2.10 creates a FL content iteratively by scanning the ARs of the SMD and expressing the variability in each iteration as follows.

1. Feature name that has a specific and meaningful name presented its role in the resulting SPL and for human uses.
2. A set of ARs that contributes to specify each feature.
3. Variability-pattern of each feature (see Variability-pattern of the feature).
4. Feature-relationship that specifies the relationship between a feature and other features (see Feature-relationship).

Step 2.11 resides a child-feature under their parent-feature.

Step 2.12 resides an 'Or' / 'Alternative' feature under their parent-feature (i.e., feature-group).

Feature-relationship The relationships between a feature and other features in a FL are defined as: a (1) parent-child relationship, like 'Mandatory' – child feature is required, 'Optional' – child feature is optional, 'Or' – at least one of the features must be selected, and 'Alternative' – one of the features must be selected. In addition, a (2) dependency-relationship (or cross-tree

constraints) are allowed, like 'requires' – the selection of 'feature A' in a product implies the selection of 'feature B', and 'excludes' – 'feature A' and 'feature B' cannot be part of the same product.

As shown in Fig. 5.7, a FL can have a set of features, where each feature has a predefined variability- pattern and feature-relationship. A feature is categorized as: a parent-feature and child-feature. Moreover, in this list, a feature is related to one or more ARs that are responsible for the feature specifications in a SMD. The documentation of features with the related requirements (i.e., ARs) enables a quick understanding of the functionality of each and explains the role of this feature in a products family.

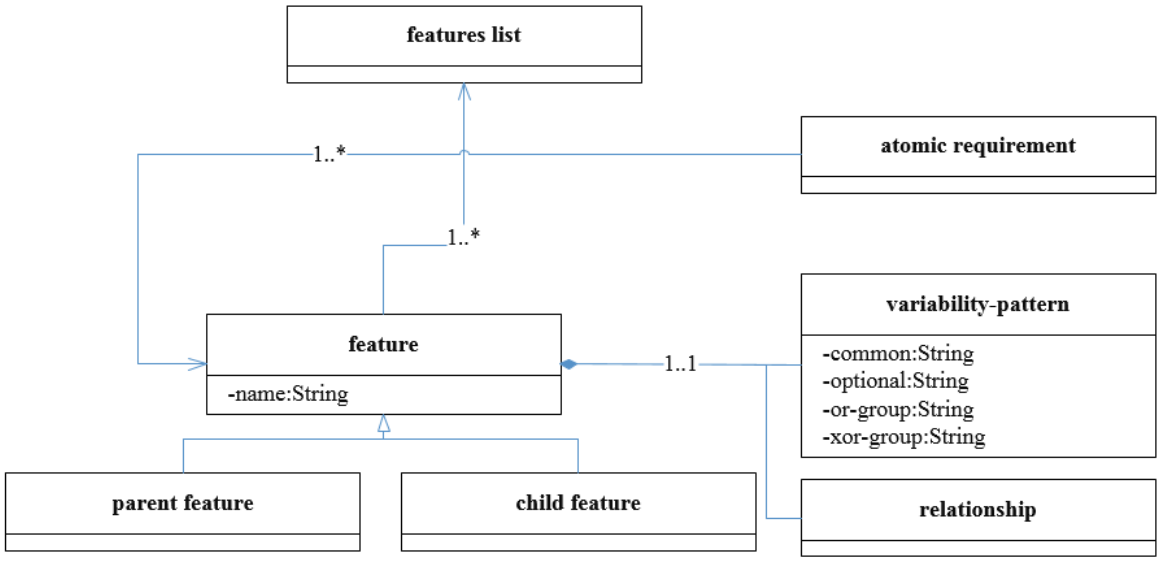


Figure 5.7: EvoSPL process: features list structure.

5.3 Feature model synthesis activity

The feature model synthesis activity constructs the current FM that represents the variability of the input documents (i.e., requirements documents) of the EvoSPL approach. In specific, this activity transforms the variability of a FL into the current FM (see Feature modelling below). To this end, the software engineers' and customers' needs are taken into consideration, since our approach uses requirements document artifacts as an input, and it allows software engineers' intervention while performing the process of the approach.

Feature modelling The current FM is a tree that represents common and variable features in the extracted SPL and is graphically described through the feature diagram notations, which specifies all products of a family through a hierarchical tree structure. The nodes of the current FM show features and edges define the relationships between features.

The activity in brief This activity (i) consumes a FL, (ii) maps the variability information from the FL to an FM and (iii) delivers the current FM that represents the initial SPL in a specific context. The final output of this activity is presented in FeatureIDE format [Thü+14]. This activity helps software engineers to manually design the current FM using FeatureIDE editor that draws, updates, and exports the model.

The feature model synthesis activity maps a FL to the current FM, to deliver a model (i.e., the current FM) that contains the following elements.

1. The feature name identifies the feature and it is put inside the respective symbol (represented by a rectangle) in the tree of the current FM.
2. The variability-pattern of a feature indicates one of the patterns that are presented in the variability-pattern of the feature (see Variability-pattern of the feature).
3. The feature-relationship that produces configurations of features defined as a combination of products work successfully under a given SPL (see Feature-relationship).

Feature model construction To construct the current FM that represents the variability found in the input requirements documents, the role of software engineers is to use FeatureIDE editor (or any proper feature modelling editor) for designing the current FM (see The current FM tree). To construct the current FM, software engineers need to perform the steps as follows (Fig. 5.8 exemplifies the steps through a simple FM 'mobile phone family'). First, software engineers (step 3.1) specify and draw the root feature of the feature diagram (see Feature diagram). Next, they (step 3.2) start with the first feature in a FL, (step 3.3) pick the feature name and draw it inside the respective symbol (rectangle) inside the feature diagram of FeatureIDE editor, and (step 3.4) define the variability-pattern of the feature.

Next, they need to (step 3.5) draw the feature's child-feature (if exists) and (step 3.6) define the feature-relationship between them. The tree of the FeatureIDE models the variability-pattern and parent-child relationship of the features. Software engineers need to perform these steps until they reach the last feature in the FL. Finally, they need to (step 3.7) use the propositional logic-method capability of the tool (see Constraint dialog) to express the dependency-relationship inside the current FM.

By the end of the feature model synthesis activity of the reverse engineering phase, the software engineers can derive the current FM that defines which feature combinations lead to valid products within the initial SPL, for the first time. Besides, they can refine and redesign the current FM in the forward engineering phase when required.

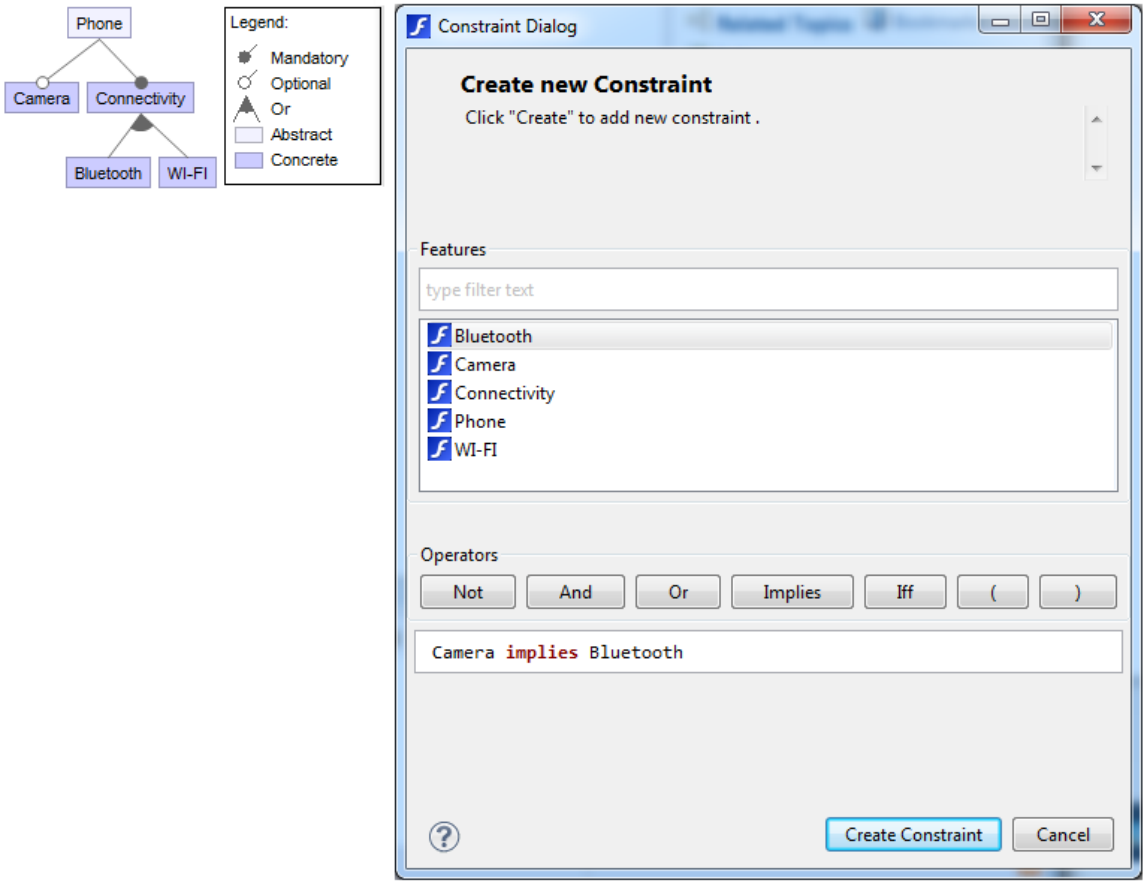


Figure 5.8: Constraints Dialog of the FeatureIDE.

The current FM tree The current FM is a tree that contains a set of features of the extracted SPL and their relationships. A feature can be decomposed into more features that are Common, Optional, Or, and Alternative. As shown in the left-hand side of Fig. 5.8, the root of the tree represents the root feature (i.e., Phone feature). First, the root feature is the main root of the tree. Hence, there is always exactly one roof feature of the tree, and it is taken from the name of the SPL ('Phone').

Typically, the remaining nodes of the tree represent a solitary feature (e.g., Camera feature), feature group (e.g., Connectivity feature), or grouped feature (e.g., Bluetooth feature). A solitary feature can be Mandatory (e.g., Connectivity feature) or Optional (e.g., Camera feature), and it can be composed of none (e.g., Camera) or more solitary features, as well as feature groups (e.g., Connectivity feature). A feature group (e.g., Connectivity feature) consists of a set of grouped features (alternative features). A set of grouped features can be contained by an 'Or' relation (e.g., Bluetooth and WI-FI features) or an 'Alternative' relation. The first one forces to choose one grouped feature or more. The second one forces to choose only one grouped feature.

However, the EvoSPL approach considers the parent-feature originated from a common parent AR, where this AR within a group of all the ARs of a SMD, and it has a parent-child relationship with all ARs of a SMD, as the root feature. Normally, the root feature appears in the first row of a SMD and FL (see Table 5.3 and Table 5.4).

Feature diagram As shown in the left-hand side of Fig. 5.8, the feature diagram is the graphical representation of the current FM. The diagram presents the root feature of the tree (e.g., Phone), features (e.g., Connectivity), variability-pattern of the feature (e.g., filled circle on the top of the rectangle), and feature-relationship (e.g., 'Connectivity' feature has a parent-child relationship with the 'Bluetooth' feature). The variability-pattern of a feature depicted as follows.

- with a filled circle at the child end of the relation for a Common (Mandatory) feature.
- with an empty circle at the child end of the relation for an Optional feature.
- as filled arcs for features forms an 'Or' relation.
- as empty arcs for features forms an 'Alternative' relation.

Besides, it also shows the overall constraints/dependency-relationship. The legend provides information about the elements used in the feature diagram.

Constraints dialog To express the constraint (dependency-relationship) inside the current FM (e.g., Camera implies Bluetooth), software engineers can formulate such constraint using the Constraint dialog of FeatureIDE. As shown in the right-hand side of Fig. 5.8, the constraints window is a text editor with auto-completion supported, auto-validity check and other tools [Git].

To create constraints inside the feature diagram that models the current FM, the Constraints dialog of FeatureIDE supports (i) a list of available features and a filtering method for this list, (ii) a list of available operators, including, Not, and Or, Implies, Iff, and parentheses (always used in pairs), (iii) a free-text editor where you can formulate constraints according to the grammar, and (iv) the dialog's control buttons where the user can save or abort a constraint.

5.4 An illustrative example of the reverse engineering phase: ATM products family

The illustrative example presented in chapter 4 will be used all over the phases of the EvoSPL approach. Thus, to preliminary illustrate the reverse engineering phase of the EvoSPL approach,

we consider two requirements document files of the ATM products family. One of the requirements documents belongs to Product 1 (initial release product) and the other one belongs to Product 3 (see R4 of Specific requirements of the difference analysis activity above). Next, we apply the steps of the reverse engineering phase on the documents, and we derive and construct the current FM (see Fig. 5.13) that models variability of the given requirements documents of the ATM products family. We explain the reverse engineering phase by following step-by-step what is presented in this chapter as below.

Difference analysis activity

Step 1.1 rewrites the requirements document of each product (Product 1 and Product 3) into ARs structure.

Step 1.2 stores the new structure of the requirement document into an RS document. The RS of Product 1 is then considered for comparison with the RS of Product3 (see Fig. 5.9).

Product 1 **RS 1: atomic requirements**	
id	**statement**
R1	Performs ATM transaction.
R2	ATM checks the balance.
R2.1	
R3	ATM withdraws the cash.
R3.1	
R4	ATM deposits the cash.
R5	
R6	ATM quits the transaction.

Product 3 **RS 3: atomic requirements**	
id	**statement**
R1	Performs ATM transaction.
R2	ATM checks the balance.
R2.1	ATM shows the balance on screen.
R3	ATM withdraws the cash.
R3.1	ATM withdraws the cash with limit.
R4	ATM deposits the cash.
R5	ATM transfers the money.
R6	ATM quits the transaction.

Figure 5.9: EvoSPL process: difference analysis activity: requirement specifications documents of Product 1 and Product 3.

Step 1.3 performs a text-based (line-based) comparison that matches the RS of each product against each other, to specify the similar and variable ARs between them inform of change terms.

The result is similar and different ARs in the matched RS documents. By the end of this step and for simplicity, the matching results are stored in the same RS document of each product (see Fig. 5.10).

Product 1 **RS 1: atomic requirements**			**Product 3** **RS 3: atomic requirements**	
id	**statement**		**id**	**statement**
R1	Performs ATM transaction.	*matched*	R1	Performs ATM transaction.
R2	ATM checks the balance.	*matched*	R2	ATM checks the balance.
R2.1		*added*	R2.1	ATM shows the balance on screen.
R3	ATM withdraws the cash.	*matched*	R3	ATM withdraws the cash.
R3.1		*added*	R3.1	ATM withdraws the cash with limit.
R4	ATM deposits the cash.	*matched*	R4	ATM deposits the cash.
R5		*added*	R5	ATM transfers the money.
R6	ATM quits the transaction.	*matched*	R6	ATM quits the transaction.

Figure 5.10: EvoSPL process: difference analysis activity: requirements specification documents with the change terms of the atomic requirements.

Variability analysis activity

Step 2.1 assigns the variability-pattern of each AR and stores them in the VD of each product. Fig. 5.11 depicts the VDs with the variability-pattern of each ARs.

Step 2.2 compares (matches) the VD of each product against each other.

Step 2.3 transfers the ‘common’ AR of the VDs into the SMD without alteration and labels the AR with a ‘common’ keyword (see Fig. 5.12).

Step 2.4 transfers the ‘optional’ AR of the VDs into the SMD and labels the AR with an ‘optional’ keyword. Furthermore, it specifies the VD and the product that the AR belongs to (see Fig. 5.12).

Product 1 **VD1 : atomic requirements**	
id	**statement**
R1 'common'	Performs ATM transaction.
R2 'common'	ATM checks the balance.
R2.1	
R3 'common'	ATM withdraws the cash.
R3.1	
R4 'common'	ATM deposits the cash.
R5	
R6 'common'	ATM quits the transaction

Product 3 **VD3 : atomic requirements**	
id	**statement**
R1 'common'	Perform ATM transaction.
R2 'common'	ATM checks the balance.
R2.1 'optional'	ATM shows the balance on screen.
R3 'common'	ATM withdraws the cash.
R3.1 'common'	ATM withdraws the cash with limit.
R4 'common'	ATM deposits the cash.
R5 'optional'	ATM transfers the money.
R6 'common'	ATM quits the transaction

Figure 5.11: EvoSPL process: variability analysis activity: variability documents with the variability-pattern of each atomic requirement.

The SMD of VD1 and VD3 **atomic requirements**	
id	**statement**
R1 'common'	Perform ATM transaction. Iteration 1: Group 1= {R1, R2, R2.1, R3, R3.1 R4, R5,R6}
R2 'common'	ATM checks the balance. Iteration 2 Group 2= {R2, R2.1}
R2.1 'optional'	ATM shows the balance on screen. VD3\Product 3
R3 'common'	ATM withdraws the cash. Iteration 3: Group 3= {R3, R3.1}
R3.1 'optional'	ATM withdraws the cash with limit. VD3\Product 3
R4 'common'	ATM deposits the cash.
R5 'optional'	ATM transfers the money. VD3\Product 3
R6 'common'	ATM quits the transaction.

Figure 5.12: EvoSPL process: variability analysis activity: single master document.

Step 2.5 transfers the 'or-group' / 'xor-group' AR into a SMD under its parent AR and labels the AR with 'or-group' / 'xor-group' keyword. Furthermore, it specifies the VD and the product that the AR belongs to (see Fig. 5.12).

Step 2.6 repeats step 2.3 to step 2.5 until the last AR of the VDs is reached.

Next, as shown in Fig. 5.12, the visualization technique capabilities 1 and 2 can be applied on the SMD as follows.

- The visualization technique colors the common and variable (different) ARs in green and yellow bars respectively. Each bar visualizes an AR and the length of the bar depends on the length of the statement. For instance, the technique colors R2 and R2.1 with green and yellow bars, respectively.
- The visualization technique uses keywords such as 'common', 'optional', 'or', and 'alternative' to define the variability-patterns of each AR. An 'or' and 'Alternative' symbolize 'or-group' and 'xor' group respectively.

Step 2.7 (text-parsing) analyses each AR in the SMD and identifies its constituents with the variability term role (see Variability term role).

Step 2.8 (naming features) suggests a proper name for the feature from the set of the extracted terms of each AR, using NFM (see Naming features method)

Step 2.9 (variability identification) identifies and organizes the variability information of the SMD, to fit its transformation into a FL (as a pre-step to feature model synthesis activity) using FIM (see Feature identification method). Table 5.3 depicts the SMD updated by FIM.

Step 2.10 creates the FL content iteratively by scanning the ARs of the SMD and expressing variability in each iteration as follows (see Table 5.4).

1. Feature name that has a specific, meaningful name presented its role in an SPL and for human uses.
2. A set of ARs that contributes to the same feature.
3. Variability-pattern of each feature (see Variability-pattern of the feature).
4. Feature-relationship that specifies the relationship between a feature and other features (see Feature-relationship).

Step 2.11 resides child-feature under their parent-feature (see Table 5.4).

Step 2.12 resides an 'Or' / 'Alternative' feature under their parent-feature (i.e., feature-group).

Listening 1:
Relationships among the ARs:

R3 'common'	ATM withdraws the cash.
R3.1 'optional'	ATM withdraws the cash with limit. VD3\Product 3

Extract (parse) the terms of each AR:
R3= {ATM, withdraws, cash}
R3.1= {ATM, withdraws, cash, limit}
Apply normalization on the terms (e.g., remove 's' from withdrawals term).
Investigate the relationships between the terms of R3 and R3.1:
R3= {ATM, withdraw, cash}
R3.1= {ATM, withdraw, cash, limit}
1. Shared terms = {ATM, withdraw, cash} being an indicator of a relationship between R3 and R3.1.
2. R3.1 explains R3.
R3 and R3.1 are related ARs.
R3 is mandatory/common parent AR
End of Listing 1

As shown in Table 5.4, the visualization technique capabilities 3,4, and 5 can be applied on the FL as follows.

- The visualization technique colors the related ARs with the same color.
- The visualization technique colors a parent-feature and its related child-feature with the same color.
- The visualization technique assigns each feature (a parent-feature and its' child-feature) the same color of ARs that contributes to this feature.

Feature model synthesis activity - Feature model construction

Table 5.3: EvoSPL process: variability analysis activity: single master document updated by the feature identification method.

The SMD of VD1 and VD3 - (atomic requirements)		
id	statement	terms\feature name\variability-pattern
R1 'common' parent AR	Perform ATM transactions.	R1 = {perform, ATM, transaction} feature name= ATM _transaction variability-pattern= 'Common' parent-feature R1 has a parent-child relationship with R2, R2. R3.1, R4, R5, R6 root feature
R2 'common' parent AR	ATM checks the balance.	R2 = {ATM, check, balance} feature name= check_balance variability-pattern= 'Common' parent-feature R2 has a parent-child relationship with R2.1
R2.1 'optional'	ATM shows the balance on screen. VD3\Product 3	R2.1 = {ATM, show, balance, screen} feature name=show_balance_screen variability-pattern= 'Optional' child-feature R2.1 has a parent-child relationship with R2
R3 'common' parent AR	ATM withdraws the cash.	R3 = {ATM, withdraw, cash} Feature name= withdrawal_ cash variability-pattern= 'Common' parent-feature R3 has a parent-child relationship with R3.1
R3.1 'optional'	ATM withdrawals the cash with a limit. VD3\Product 3	R3.1= {ATM, withdraw, cash, limit} Feature name= withdrawal_ cash_ limit variability-pattern= 'Optional' child-feature R3.1 has a parent-child relationship with R3
R4 'common'	ATM deposits the cash.	R4 = {ATM, deposit, cash} Feature name= deposit_cash variability-pattern= 'Common' child-feature R4 has a parent-child relationship with R1
R5 'optional'	ATM transfers the money. VD3\Product 3	R5= {ATM, transfer, money} Feature name= transfer_money variability-pattern= 'Optional' child-feature R5 has a parent-child relationship with R1
R6 'common'	ATM quits the transaction.	R6= {ATM, quit, transaction} Feature name= quit_transaction variability-pattern= 'Common' child-feature R6 has a parent-child relationship with R1

Table 5.4: EvoSPL process: variability analysis activity: features list.

Features list
Root feature: ATM transaction
Feature 1: name= check_balance: {R2, R2.1} variability-pattern= Common parent-child relationship with R1 child-feature: Feature 2
Feature 2: name= show_balance_screen: {R2.1} variability-pattern= Optional parent-child relationship with R2 child-feature: none
Feature 3: name= withdraw_cash: {R3, R3.1} variability-pattern= Common parent-child relationship with R1 child-feature: Feature 4
Feature 4: name= withdraw_cash_limit: {R3.1} variability-pattern= Optional parent-child relationship with R3 child-feature: none
Feature 5: name= deposit_cash: {R4} variability-pattern= Common parent-child relationship with R1 child feature: none
Feature 6: feature name= transfer_money: {R5} variability-pattern= Optional parent-child relationship with R1 child-feature: none
Feature 7: name= quit_transaction: {R6} variability-pattern= Common parent-child relationship with R1 child feature: none

To construct the current FM that represents the variability found in the input requirements documents of Product 1 and Product 3, as depicted in Fig. 5.13, the role of software engineers is to use FeatureIDE editor for designing the current FM. To construct the current FM, software engineers need to (step 3.1) specify and draw the root feature ('ATM_ transaction') of the feature diagram. Next, they (step 3.2) start with the first feature in the FL ('check_balance'), (step 3.3) pick the feature name and draw it inside the respective symbol (rectangle) inside the feature diagram (of FeatureIDE editor, and they (step 3.4) define the variability-pattern of the feature (Common - filled circle on the top of the rectangle).

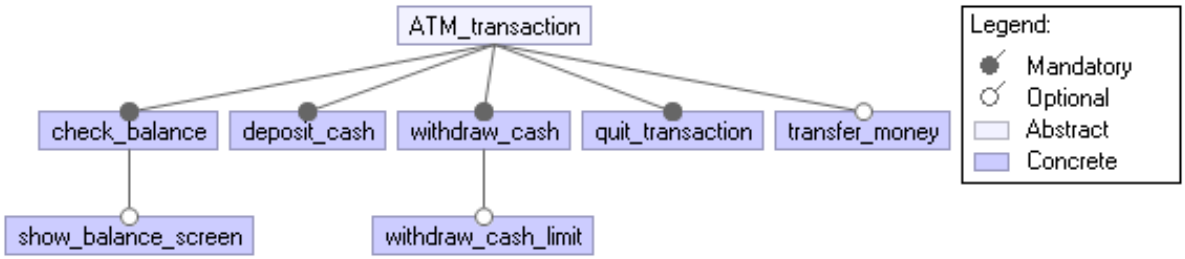


Figure 5.13: EvoSPL process: feature model synthesis activity: the current FM.

Next, they need to (step 3.5) draw its' child-feature ('show_balance_screen') and (step 3.5) define the feature-relationship between them ('check_balance' feature has a parent-child relationship with the 'show_balance_screen'). The tree of the FeatureIDE models the variability-pattern and parent-child relationship of the features. Software engineers need to perform these steps until they reach the final feature ('quit_transaction') in the FL. Finally, they need to (step 3.7) use the propositional logic-method capability of the tool to express the dependency-relationship inside the current FM. This example does not consider the dependency-relationship between the features of the current FM. Thus, we can ignore the final step.

Chapter 6

EvoSPL : Forward engineering phase

The adoption of an SPL requires (1) analysing commonality and variability of the existing artifacts and (2) tracking the changes that were introduced to these artifacts. As introduced earlier, the reverse engineering phase supports what is required in (1), and this chapter presents the forward engineering phase that contributes to support what is required in (2), when adopting an SPL practice. Thus, the forward engineering phase follows the reverse engineering phase and allows to transform the remaining set of products of a family into the already extracted SPL, namely the initial SPL. Furthermore, this phase contributes to adapting the bootstrapped SPL with a new product, to deliver the resulting SPL.

Fig. 6.1 presents the forward engineering phase that is subdivided into two activities: the (1) bootstrapping and (2) evolution. The activities provide steps to avoid the problems that might occur when evolving SPLs. Both activities use the feature model refactoring scenario to refine the current FM when required. The scenario adopts a set of refactoring notions that are important for safely evolving the existing SPL (the initial, bootstrapped, or resulting SPL) by simply improving its design or even by adding (new) products while preserving the existing ones.

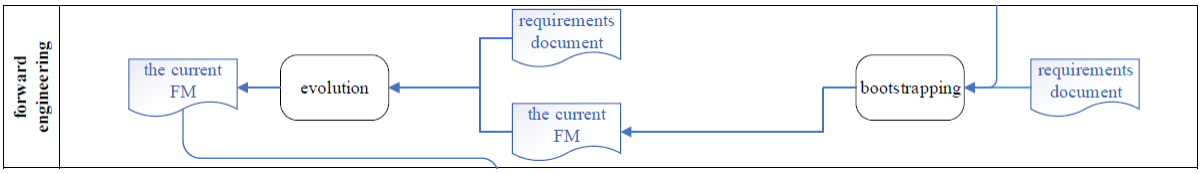


Figure 6.1: EvoSPL process: forward engineering phase.

6.1 Bootstrapping activity

Evolving the initial SPL with the requirements of one product of a family requires to consider both the initial SPL and the variability of the artifacts. The requirements of a product need to be propagated to the current FM and then to the initial SPL, which leads to keeping the consistency between them. Thus, the bootstrapping activity supports adding the remaining set

of products family into the already extracted SPL (i.e., the initial SPL), one by one. Hence, it requires to refine the current FM with variability information (features) of the selected product, according to domain experts' decision or request.

The activity in brief The bootstrapping activity (i) uses the requirements document of the selected product of a family, (ii) identifies the features of this product, and (iii) stores them in a FL. Finally, it (iv) evolves the current FM with features stored in the FL of the product using the feature model refactoring scenario (section 6.3). This activity adopts the bootstrapping steps to perform the bootstrapping process.

The bootstrapping steps Using one product of a family, this activity consumes the requirement document of the product (one of the remaining set of a product family) at once, to apply the bootstrapping steps that are defined for the EvoSPL approach as follows.

Step 1.1 rewrites the requirements document of the product into a set of ARs (see Atomic requirement structure of reverse engineering phase in Chapter 5).

Step 1.2 stores the new structure of the requirement document in an RS document (see RS preparation of reverse engineering phase in Chapter 5). The RS is then considered for comparison with the SMD of a product family.

Step 1.3 performs a text-based (line-based) comparison that matches the RS of the product against the SMD, to specify the similar and variable ARs of the product (see Change terms of the reverse engineering phase in Chapter 5). The result is similar and different ARs between the matched documents (distance between the documents). By the end of this step and for simplicity, the matching results are stored in the same VD of the product.

Step 1.4 assigns the variability-pattern of each AR (see Variability-pattern of the AR of the reverse engineering phase in Chapter 5) and stores them in a VD of the product.

Step 1.5 applies the visualization technique capabilities 1, 2, and 3 (see Visualization technique macro step of the reverse engineering phase in Chapter 5) on the VD of the product.

Step 1.6 applies the feature identification macro step (see Feature identification macro step of the reverse engineering phase in Chapter 5) on the VD of the product.

Step 1.7 applies the variability transformation macro step (see Variability transformation macro step of the reverse engineering phase in Chapter 5) on the VD of the product.

Step 1.8 applies the feature model refactoring scenario (section 6.3), to refine the current FM with features of the product stored in the FL of this product.

The steps are repeated until all the products of a family are considered, and the bootstrapped SPL is delivered. Hence, the bootstrapping activity involves bootstrapping a set of products that need to be added to the initial SPL to deliver the bootstrapped SPL. Hence, the bootstrapping activity may use the bootstrapping steps many times to restructure the initial SPL to encompass the (remaining) set of the products (one by one).

6.2 Evolution activity

In the EvoSPL, a part of its evolution-based process is to adapt the bootstrapped and resulting SPL, including its related FM (i.e., the current FM) with new requirements that would modify its structure (e.i., introduce a new feature or functionality to the current FM), while the functionality and capability of the existing SPL are not affected. The EvoSPL approach includes the evolution activity to handle a continuous refinement of the resulting SPL (already extracted and bootstrapped using our approach) based on the changes done on the current FM in a way that has no impact on the validity of the resulting products. The evolution activity (1) evolves the bootstrapped SPL to encompass a new product, by refining the current FM with the requirements (features) of the new product and delivers the resulting SPL. The evolution activity (2) evolves the resulting SPL with another new product whenever required.

The activity in brief The evolution activity (i) uses the requirements document of a new product upon receiving a new customer request, (ii) identifies features of the new product, and (iii) stores them in a FL. Finally, it (iv) evolves the current FM with features of the new product listed in a FL, using the feature model refactoring scenario (section 6.3).

The evolution steps The evolution activity consumes the requirement document of a new product, to apply the evolution steps that are defined for the EvoSPL approach as follows.

Step 2.1 rewrites the requirements document of the new product into a set of ARs (see Atomic requirement structure of reverse engineering phase in Chapter 5).

Step 2.2 stores the new structure of the requirement document in an RS document (see RS preparation of reverse engineering phase in Chapter 5). The RS is then considered for comparison with the SMD of a product family.

Step 2.3 performs a text-based (line-based) comparison that matches the RS of the new product against the SMD, to specify the similar and variable ARs of the new product (see Change terms

of the ARs of reverse engineering phase in Chapter 5). The result is similar and different ARs between the matched documents (distance between the documents). By the end of this step and for simplicity, the matching results are stored in the same RS of the new product.

Step 2.4 assigns the variability-pattern of each AR (see Variability-pattern of the AR of reverse engineering phase in Chapter 5) and stores them in a VD.

Step 2.5 applies the visualization technique capabilities 1, 2, and 3 (see Visualization technique macro step of reverse engineering phase in Chapter 5) on the VD of the new product.

Step 2.6 applies the feature identification macro step (see Feature identification macro step of reverse engineering phase in Chapter 5) on the VD of the new product.

Step 2.7 applies the variability transformation macro step (see Variability transformation macro step of reverse engineering phase in Chapter 5) on the VD of the new product.

Step 2.8 applies the feature model refactoring scenario (section 6.3), to refine the current FM with the features of the new product stored in the FL of the new product.

It is worth a mention that the evolution activity steps can be used whenever it is required to encompass the resulting SPL with a new product.

6.3 Feature model refactoring scenario

The bootstrapping activity and the evolution activity uses the feature model refactoring scenario designed for the EvoSPL approach. The scenario applies predefined steps to refine an FM (i.e., the current FM) with the features of a (new) product. The scenario adopts a set of sound refactoring for FMs (see Feature model refactoring below) to perform the transformations that guarantees configurability improvement of the current FM. The feature model refactoring scenario addresses a set of steps required when evolving the existing SPL with features of a (new) product with a focus on FMs refactoring.

On the level of a products family, (new) products can be added. Basically, adding (new) products should not require any changes to the other products and the SPL. However, in practice, a specific type of changes of requirements (features) in the products of an SPL and their impact require to promote those changes to a variability model (i.e., the current FM) and then to the SPL level. For instance, common functionality (feature) can be made variable if it should be excluded from some products. Thus, the feature model refactoring scenario uses FMs refactoring

to refine the current FM with (new) requirements of (new) product(s) that are not supported by the existing SPL. For example, a new common feature can be added to the current FM only if it represents a common functionality among all the products of a family and a common feature can be converted to an optional feature if it no longer appears in all the products.

However, the feature model refactoring scenario uses a sequence of FMs (namely, the current FM is refined many times when required upon the changes occur) to keep track of modifications to the existing SPL. The feature model refactoring scenario consists of small steps (see Feature model refactoring steps below) that may lead to changes in the existing SPL, resulting in features being added to or eliminated from the existing SPL. The EvoSPL approach proposes this scenario to keep the existing SPL and its products consistent with the current FM. The scenario adopts various refactoring notations applicable to FMs (i.e., the current FM). Additionally, it adopts several changes include adding a new feature, eliminating a feature, or changing a common feature to an optional feature and vice versa.

Feature model refactoring steps The EvoSPL approach adopts the SPL refactoring process that may affect only the current FM and consider refactoring of a single FM at once. So far, as described before, the variability analysis activity deals with the requirements document of a (new) product to derive its features and create a FL that contains the features of the (new) product. The feature model refactoring scenario merges the features of the (new) product to the current FM. Thus, to import a (new) product into the existing SPL, the feature model refactoring scenario adopts the following steps.

Step 1 reads a feature (top to down) from the FL and compares the feature to nodes of the current FM.

Step 2 matches this feature including its variability information (feature name and variability-pattern) with nodes of the current FM. The model is traversed to identify a node with a match as well as with a different variability information.

Step 3 checks the result of the match status. The feature model refactoring scenario considers the following status.

Status (1): a feature appears Optional in the current FM and, at the same time, appears Common in the FL of a (new) product. This leads to keeping the current FM unchanged.

Status (2): a feature appears in the current FM with the same variability information. This leads to keeping the current FM unchanged. So, the refactoring notations are only applied in case of differences in the variability information between a feature and the current FM.

Table 6.1: Feature changes in the current feature model.

feature changes in the current FM	
1.	Add a new Common feature to the current FM.
2.	Add a new Optional feature to the current FM.
3.	Add a new Or (or-group) feature to the current FM.
4.	Add a new Alternative (xor-group) feature to the current FM.
5.	Remove a feature from the current FM (see Removing a feature).
6.	Convert Optional feature into Common feature in the current FM.
7.	Converting Or (or-group) into Alternative (xor-group).
8.	Converting Alternative (xor-group) into Or (or-group).

Status (3): a feature requires a change to the current FM (see Table 6.1). This leads to applying proper refactoring notations (see A catalog of sound FM refactoring below) to the current FM.

Feature changes The feature model refactoring scenario of the EvoSPL approach proposes a list of feature changes to evolve the current FM. As shown in Table 6.1, it defines eight types of changes. However, it is possible to define other types of feature changes. Our approach can be easily extended to handle new types of feature changes.

Removing a feature Removing a feature from the current FM, a situation where the upcoming new products no longer use this feature. Thus, this feature is a good candidate to remove from the current FM. Thus, by removing a feature from the current FM, the artifacts related to that feature, such as implemented modules, the reference architecture, feature mapping, and related code should be changed in such a way that the consistency of the resulting SPL remains intact. Removing or adding a feature to the current FM may include or exclude some pieces of code, especially whenever a feature is linked to some code fragments throughout feature mapping. Our approach supports a tool that updates such feature mapping, based on the changes occurring to the fearers (section 7.3).

Feature model refactoring The EvoSPL approach adopts an FM refactoring that involves a transformation, which improves the quality of the current FM, by maintaining or increasing its configurability. For example, consider the current FM at point1 and point 2. So according to the definition of an FM refactoring that is adopted by our approach, the current FM at point2 refactors the current FM at point1 if and only if all valid configurations of the current FM at point1 are valid configurations of the current FM at point2. Fig. 6.2 depicts the current FM at two points. It describes the language of an ATM machine. In the left-hand side, the ATM supports Arabic or English language. Suppose that we would like to refactor the left-hand side model (the current FM at point 1) to the right-hand side model (the current FM at point 2), by adding a new alternative. So, we can have an additional language (Portuguese) in the resulting model, while still maintaining the earlier configurations.

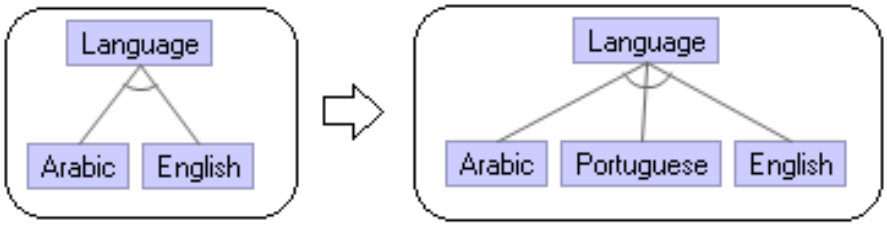


Figure 6.2: Feature model refactoring example.

The left-hand side model has two valid configurations: {Language, Arabic},{ Language, English}. The right-hand side model has the same configurations of the left-hand side model plus the {Language, Portuguese} configuration. This ensures correctness of the refactoring depicted in Fig. 6.2, since the right-hand side model improves the configurability of the LHS model. To apply such FM refactoring on the current FM of the EvoSPL whenever required, we present a catalog of sound FM refactoring (see A catalog of sound FM refactoring).

A catalog of sound FMs refactoring The feature model refactoring scenario depends on a catalog of sound FM refactoring proposed by Alves et al. [Alv+06b]. According to the definition of SPL refactoring, a catalog involves FM refactoring and offers a number of sound unidirectional and bidirectional FM refactoring. As shown in Fig. 6.3, each refactoring consists of two templates (patterns) of FMs, one on the left-hand side and the other on the right-hand side. The feature model refactoring scenario applies a refactoring whenever the left template is matched by a given (part of) the current FM. The refactoring template only shows the differences between the FMs. Moreover, a dashed line on top of a feature indicates that this feature may have a parent feature. A dashed line below a feature indicates that this feature may have additional sub-features. The adopted catalog of refactoring by our approach is summarized in Appendix A.

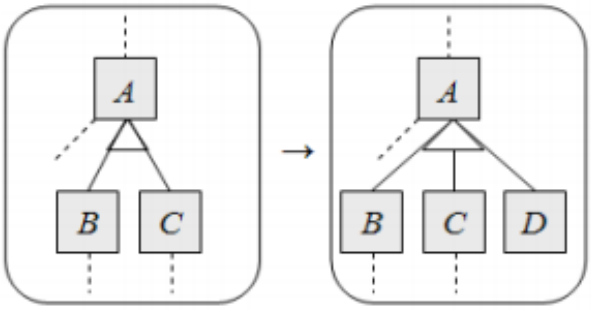


Figure 6.3: Refactoring 5 - catalog of sound feature model refactoring [Alv+06b].

Unidirectional refactoring Table 6.2 presents some FM refactoring in order to refine the current FM whenever the changes occur. For instance, Refactoring 5 (see Fig. 6.3) allows adding new feature D to an xor-group to increase the alternatives between B and C. This refactoring is

Table 6.2: Summary of unidirectional feature model refactoring.

Refactoring	Name	Refactoring	Name
1	Convert Alternative to Or	7	Convert Mandatory to Optional
2	Collapse Optional and Or	8	Convert Alternative to Optional
3	Collapse Optional and Alternative to Or	9	Pull up node
4	Add Or between Mandatory	10	Push down node
5	Add new Alternative	11	Remove formula
6	Convert Or to Optional	12	Add Optional node

the general version of the specific refactoring that can be applied in cases, such as the specific refactoring shown in Fig. 6.2 The feature model refactoring scenario uses Refactoring 5 to the specific models depicted in Fig. 6.2 by matching the variables A, B, C and D with the specific features Language, Arabic, English, and Portuguese, respectively. Note that there is no dashed line below D in the right-hand side of this refactoring because it only introduces a new feature without sub-features. The other lines of right-hand side are necessary to preserve the feature matched by the dashed line on the left-hand side.

Refactoring 5 is sound because the resulting model contains all the configurations from the original one, also allowing a configuration containing A and D in the absence of B and C. Hence, this transformation improves a model by increasing its configurability. It is worth mentioning that all the refactoring presented in Table 6.2 (see Appendix A.1 also) can be applied from right to left but not the reverse. Besides, most of them can be applied similarly in context with more than two features, such as Refactoring 2 and Refactoring 5.

Bidirectional refactoring Table 6.3 presents a set of bidirectional refactorings (B- Refactorings) for FMs. A bidirectional refactoring is a special case of FM refactoring that maintain the configurability of a model. Theoretically, if two FMs have the same configurability (semantics), we can always relate them by applying B-Refactoring. It also defines two FM templets, although being applicable in both directions. For example, B-Refactoring 1 relates Alternative and Or relations, it and works from left to right allows converting Alternatives to Or relation along with two formulas establishing the same constraints. Similarly, by applying the transformation from right to left, it allows converting Or to Alternative (see Appendix A.2).

The root feature of an FM always appeared in all valid configurations. Practically, some of the previous transformations may not be useful, since they convert a valid FM to another that is not a tree, such as B-Refactoring 4. They are important for theoretical reasoning. In practice, developers should only be aware of the FM refactoring catalog shown in Table 6.2 (see Unidirectional refactoring).

Table 6.3: Summary of bidirectional feature model refactoring.

B-Refactoring	Name
1	Replace Alternative
2	Replace Or
3	Replace Mandatory
4	Replace Optional

This chapter presents only some B-refactoring. The list of all such refactorings and a proof that they are complete can be found in [Alv+06a]. The completeness proof shows that, for any two semantically equivalent FMs, there is a strategy consisting of consecutive applications of B-refactoring such that it transforms one FM to another. This result means their refactorings are sufficiently expressive for the FM language that we consider in the EvoSPL approach.

Properties of the catalog The EvoSPL approach chooses the catalog of sound refactorings in order to apply an FM refactoring, since the catalog (1) can be seen as a high-level API, which is much easier to use (based on template matching). Semantics for FMs are encoded in the Prototype Verification System (PVS), which is a formal specification language. Using the PVS theorem prover, all the proposed refactorings are proved with respect to a formal semantics [Ghe+06]. We think that using the PVS (2) gives a chance for proposing other FM refactoring. Both, the formalization and proofs that are guaranteed in the catalog of sound refactoring, are important in order (3) to increase the reliability when refactoring SPLs, as presented in the illustrative example and case study.

6.4 An illustrative example of the forward engineering phase: ATM products family

This section continued with the illustrative example explained in the earlier chapters, to clarify the forward engineering phase. The example combines the bootstrapping and evolution activities. Using the ATM products family, this section started from a scenario in which we derived the current FM from two products (Product 1-initial release and Product 3) in the reverse engineering phase (section 5.4). Now, we have another product of the family and a new one needs to be developed in the near future. From this initial scenario, we have two goals: 1) to bootstrap the existing product (Product 2) into the initial SPL and 2) to react the bootstrapped SPL to encompass a new product (ProductNew) of the family, to deliver the resulting SPL. Both goals involve refining the current FM with features of the new (product), which requires the refactoring notation for SPL in terms of FMs.

To provide a clear picture of the example, this section presents a sequence of the current FM at different points as follows.

Point 1. The current FM that is already derived and constructed, using two products of the ATM products family, namely Product 1 and Product 3 (section 5.4).

Point 2. The current FM that is refined with the features of a product of the ATM products family, namely Product 2.

Point 3. The current FM that is refined with the features of a new product planned to be added to the ATM products family, namely ProductNew. Point 2 and Point 3 show the current FM as a sequence of refactoring points based on the features of the (new) product, which determines the refactoring opportunities of the current FM.

6.4.1 The current FM at point 1

Point 1 shows the current FM that is derived by the reverse engineering phase using the requirements documents of two products (Produc1 and Product 3). Fig. 5.13 depicts the current FM at this point.

6.4.2 The current FM at point 2: bootstrapping activity

Using Product 2 of the ATM products family, the bootstrapping activity consumes the requirement document file of the product (one of the remaining set of the ATM product family), to apply the bootstrapping steps that are defined for the EvoSPL approach as follows.

Step 1.1 rewrites the requirements document of Product 2 into a set of ARs.

Step 1.2 stores the new structure of the requirement document in an RS document. The RS of Product 2 shown in Fig. 6.4 is then considered for comparison with the SMD of the ATM products family (see Fig. 6.5).

Step 1.3 performs a text-based (line-based) comparison that matches the RS of Product 2 against the SMD of the ATM products family, to specify the similar and variable ARs of the product. The result is similar and different ARs between the matched documents. By the end of this step and for simplicity, the matching results are stored in the same RS of Product 2, as shown in Fig. 6.5.

Product 2 **RS 2: atomic requirements**	
id	**statement**
R1	Performs ATM transaction.
R2	ATM checks the balance.
R2.1	
R3	ATM withdraws the cash.
R3.1	
R4	ATM deposits the cash.
R5	
R6	ATM quits the transaction.
R7	ATM converts between the currencies.

Figure 6.4: EvoSPL process: bootstrapping activity: requirements specification document of Product 2.

Step 1.4 assigns the variability-pattern of each AR and stores them in the VD of Product 2. Fig. 6.6 depicts the VD with the variability-pattern of each AR.

Step 1.5 applies the visualization technique capabilities 1 and 2 on the VD of Product 2. The VD looks as shown in Fig. 6.7.

Step 1.6 applies the feature identification macro step on the VD of Product 2. The VD looks as shown in Table 6.4.

Step 1.7 applies the variability transformation macro step on the VD of Product 2 and organizes it into a FL in terms of features. The FL of Product 2, which delivers as a result of applying the variability transformation macro step and the visualization technique capabilities 3, 4, and 5, looks as shown in Table 6.5 .

Step 1.8 applies the feature model refactoring scenario, to refine the current FM with features of Product 2. This step involves bootstrapping Product 2 into the initial SPL (to deliver the

Table 6.4: EvoSPL process: bootstrapping activity: variability document of Product 2 updated by feature identification method.

Product 2 : VD 2: atomic requirements		
id	statement	terms\feature name\variability-pattern
R1 'common'	Performs ATM transaction.	R1 = {perform, ATM, transaction} feature name= ATM_transaction variability-pattern= 'Common' R1 has a parent-child relationship with R2, R3, R4, R6,R7 root feature
R2 'common'	ATM checks the balance.	R2 = {ATM, check, balance} feature name= check_balance variability-pattern= 'Common' child-feature R2 has a parent-child relationship with R1
R2.1 'optional'		
R3 'common'	ATM withdraws the cash.	R3 = {ATM, withdraw, cash} feature name= withdraw_cash variability-pattern= 'Common' child-feature R3 has a parent-child relationship with R1
R5 'optional'		
R6 'common'	ATM quits the transaction.	R6= {ATM, quit, transaction} feature name= quit_transaction variability-pattern= 'Common' child-feature R6 has a parent-child relationship with R1
R7 'optional'	ATM converts between the currencies.	R7= {ATM, convert, currencies} feature name= convert_currency variability-pattern= 'Optional' child-feature R7 has a parent-child relationship with R1

Table 6.5: EvoSPL process: bootstrapping activity: features list of Product 2.

Features list of Product 2
Root feature: ATM transaction
Feature 1: name= check_balance: {R2} variability-pattern= Common parent-child relationship with R1 child feature: none
Feature 2: name= withdraws_cash: {R3} variability-pattern= Common parent-child relationship with R1 child feature: none
Feature 3: name= deposit_cash: {R4} variability-pattern= Common parent-child relationship with R1 child feature: none
Feature 4: name= quits_transaction: {R6} variability-pattern= Common parent-child relationship with R1 child-feature: none
Feature 5: name= convert_currency: {R7} variability-pattern= Optional parent-child relationship with R1 child-feature: none

Product 2 **RS 2: atomic requirements**			**The SMD of VD1 and VD3** **atomic requirements**	
id	**statement**		**id**	**statement**
R1	Performs ATM transaction.	***matched***	R1	Perform ATM transaction.
R2	ATM checks the balance.	***matched***	R2	ATM checks the balance.
R2.1		***deleted***	R2.1	ATM shows the balance on screen.
R3	ATM withdraws the cash.	***matched***	R3	ATM withdraws the cash.
R3.1		***deleted***	R3.1	ATM withdraws the cash with limit.
R4	ATM deposits the cash.	***matched***	R4	ATM deposits the cash.
R5		***deleted***	R5	ATM transfers the money.
R6	ATM quits the transaction.	***matched***	R6	ATM quits the transaction.
R7	ATM converts between the currencies.	***added***	R7	

Figure 6.5: EvoSPL process: bootstrapping activity: requirements specification of Product 2 and the SMD with the change terms of each atomic requirement.

bootstrap SPL) using the feature model refactoring scenario. The tracing of the feature model refactoring scenario steps is presented in Table 6.6.

Refactoring point 1.1 of the current FM The 'convert_currency' feature appears Optional in the FL of Product 2 and does not exist in the current FM of the ATM products family. This leads to add 'convert_currency' feature to the current FM using Refactoring 12, which allows to add new Optional feature to the current FM. Fig. 6.8 shows the current FM in refactoring point 1.1 that is refined using the feature model refactoring scenario, in the bootstrapping activity. Red asterisk is placed next to the refactoring point to indicate an alteration in the current FM. However, this leads to propagating the feature changes from Product 2 to the current FM and then to the initial SPL.

6.4.3 The current FM at point 3: evolution activity

The evolution activity consumes the requirement document of ProductNew, to apply the evolution steps that are defined for the EvoSPL approach as follows.

Step 2.1 rewrites the requirements document of the ProductNew into a set of ARs (see Fig. 6.9).

Table 6.6: Tracing of the feature model refactoring scenario steps during the bootstrapping activity.

Step	Feature variability information	Status
Step 1 Step 2 Step 3	Feature 1: name= check_balance variability-pattern= Common	Status 2 Leads to keep the current FM unchanged.
Step 1 Step 2 Step 3	Feature 2: name= withdraw_cash variability-pattern= Common	Status 2 Leads to keep the current FM unchanged.
Step 1 Step 2 Step 3	Feature 3: name= deposit_cash variability-pattern= Common	Status 2 Leads to keep the current FM unchanged.
Step 1 Step 2 Step 3	Feature 4: name= quit_transaction variability-pattern= Common	Status 2 Leads to keep the current FM unchanged.
Step 1 Step 2 Step 3	Feature 5: name= convert_currency variability-pattern= Optional	Status 3 A feature requires a change to the current FM (see Table 6.2) The feature 'convert_currency' requires 'add a new Optional feature to the current FM (see Refactoring point 1.1 of the current FM)
Step 4	The final feature in the FL 'convert_currency' of Product 2 is reached.	
Step 5	There is no feature that appears common in the current FM and does not appear in the FL of Product 2.	

Product 2 **VD 2: atomic requirements**	
id	**statement**
R1 'common'	Performs ATM transaction.
R2 'common'	ATM checks the balance.
R2.1 'optional'	
R3 'common'	ATM withdraws the cash.
R3.1 'optional'	
R4 'common'	ATM deposits the cash.
R5 'optional'	
R6 'common'	ATM quits the transaction.
R7 'optional'	ATM converts between the currencies.

Figure 6.6: EvoSPL process: bootstrapping activity: variability document of Product 2 with the variability-pattern of each atomic requirement.

Step 2.2 stores the new structure of the requirement document in an RS document. The RS of NewProduct is then considered for comparison with the SMD of the ATM products family, as shown in Fig. 6.10.

Step 2.3 performs a text-based (line-based) comparison that matches the RS of NewProduct against the SMD of the ATM products family, to specify the similar and variable ARs of the product. The result is similar and different ARs between the matched documents. By the end of this step and for simplicity, the matching results are stored in the same RS of NewProduct, as shown in Fig. 6.10.

Step 2.4 assigns the variability-pattern of each AR and stores them in the VD of NewProduct. Fig. 6.11 depicts the VD with the variability-pattern of each AR.

Step 2.5 applies the visualization technique capabilities 1 and 2 on the VD of NewProduct. The VD looks as shown in Fig. 6.12.

Product 2 **VD 2: atomic requirements**	
id	**statement**
R1 'common'	Performs ATM transaction.
R2 'common'	ATM checks the balance.
R2.1 'optional'	
R3 'common'	ATM withdraws the cash.
R3.1 'optional'	
R4 'common'	ATM deposits the cash.
R5 'optional'	
R6 'common'	ATM quits the transaction.
R7 'optional'	ATM converts between the currencies.

Figure 6.7: EvoSPL process: applying the visualization technique on the variability document of Product 2.

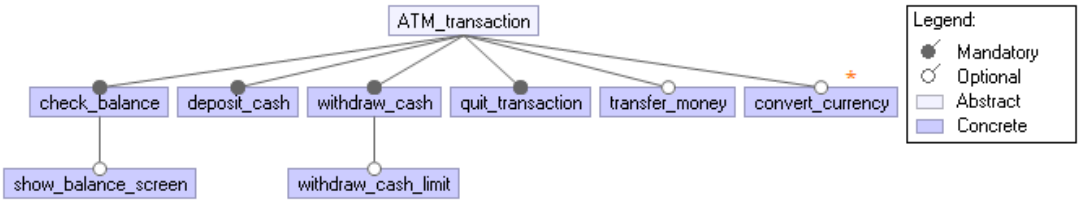


Figure 6.8: The current FM of the ATM products family in refactoring point 1.1.

Step 2.6 applies the feature identification macro step on the VD of ProductNew.

Step 2.7 applies the variability transformation macro step on the VD of NewProduct and organizes it into a FL in terms of features. The FL of NewProduct delivered as a result of applying the variability transformation macro step and the visualization technique capabilities 3, 4, and 5 looks as shown in Table 6.7.

Step 2.8 applies the feature model refactoring scenario, to refine the current FM with features

Table 6.7: EvoSPL process: evolution activity: features list of ProductNew.

Features list of ProductNew	
Root feature: ATM transaction	
Feature 1: feature name= check_balance: {R2, R2.1, R2.2} variability-pattern= Common parent-child relationship with R1 child-feature: Feature 2 and Feature 3	Feature 6: name= deposit_cash: {R4} variability-pattern= Common parent-child relationship with R1 child feature: none
Feature 2: name= show_balance_sceen: {R2.1} variability-pattern= Common parent-child relationship with R2 child-feature: none	Feature 7: name= transfer_money: {R5} variability-pattern= Common parent-child relationship with R1 child-feature: none
Feature 3: name= show_balance_report: {R2.2} variability-pattern= Optional parent-child relationship with R2 child-feature: none	Feature 8: name= quit_transaction: {R6} variability-pattern= Common parent-child relationship with R1 child-feature: none
Feature 4: name= withdraw_cash: {R3, R3.1} variability-pattern= Common parent-child relationship with R1 child-feature: Faeture 5	Feature 9: name= pay_service: {R8} variability-pattern= Optional parent-child relationship with R1 child-feature: none
Feature 5: name= withdraw_cash_limit: {R3.1} variability-pattern= Common parent-child relationship with R3 child-feature: none	

ProductNew **VDNew: atomic requirements**	
id	**statement**
R1	Performs ATM transaction.
R2	ATM checks the balance.
R2.1	ATM shows the balance on screen.
R2.2	ATM shows the balance on report.
R3	ATM withdraws the cash.
R3.1	ATM withdraws the cash with limit.
R4	ATM deposits the cash.
R5	ATM transfers the money.
R6	ATM quit the transaction
R7	
R8	ATM pays for the services.

Figure 6.9: EvoSPL process: evolution activity: requirement specification of ProductNew

of ProductNew. This step involves adding ProductNew to the bootstrapped SPL, to deliver the resulting SPL, using the feature model refactoring scenario. The tracing of the feature model refactoring scenario steps is presented in Table 6.8

Refactoring point 2.1 of the current FM The 'show_balance_report' feature appears Optional in the FL of ProductNew, and it does not exist in the current FM of ATM products family. This leads to add 'show_balance_report' feature from the FL to the current FM, using Refactoring 12. Fig. 6.13 shows the current FM in refactoring point 2.1 that is refined using the feature model refactoring scenario, in the evolution activity. Red asterisk are placed next to the refactoring points, to indicate alterations in the current FM. Furthermore, as shown in Fig. 6.14, the current FM at refactoring point 2.1.1 refactors the current FM at refactoring point 2.1 by replace Optional to Or (B-Refactoring 2).

Table 6.8: Tracing of the feature model refactoring scenario steps during the evolution activity.

Step	Feature variability information	Status
Step 1 Step 2 Step 3	Feature 1: name= check_balance variability-pattern= Common	Status 2 Leads to keep the current FM unchanged.
Step 1 Step 2 Step 3	Feature 2: name= show_balance_sceen variability-pattern= Common	Status 3 A feature requires a change to the current FM (see Table 6.2). The feature 'show_balance_report' requires add a new Optional feature to the current FM (see Refactoring point 2.1 of the current FM).
Step 1 Step 2 Step 3	Feature 3: name= show_balance_report variability-pattern= Optional	Status 3 A feature requires a change to the current FM (see see Table 6.2). The feature 'show_balance_report' requires add a new Optional feature to the current FM (see Refactoring point 2.1 of the current FM).
Step 1 Step 2 Step 3	Feature 4: name= withdraw_cash variability-pattern= Common	Status 2 Leads to keep the current FM unchanged.
Step 1 Step 2 Step 3	Feature 5: name= withdraw_cash_limit variability-pattern= Common	Status 1 Leads to keep the current FM unchanged.
Step 1 Step 2 Step 3	Feature 6: name= deposit_cash variability-pattern= Common	Status 2 Leads to keep the current FM unchanged.
Step 1 Step 2 Step 3	Feature 7: name= transfer_money variability-pattern= Common	Status 1 Leads to keep the current FM unchanged.
Step 1 Step 2 Step 3	Feature 8: name= pay_service variability-pattern= Optional	Status 3 A feature requires a change to the current FM (see see Table 6.2). The feature 'pay_service' requires add a new Optional feature to the current FM (see Refactoring point 2.2 of the current FM)
Step 4	The final feature in the FL 'pay_service' of Product New is reached.	
Step 5	There is no feature that appears common in the current FM and does not appear in the FL of Product New.	

ProductNew **RS New: atomic requirements**			**The SMD of VD1 and VD3** **atomic requirements**	
id	**statement**		**id**	**statement**
R1	Performs ATM transaction.	*matched*	R1	Perform ATM transaction.
R2	ATM checks the balance.	*matched*	R2	ATM checks the balance.
R2.1	ATM shows the balance on screen.	*matched*	R2.1	ATM shows the balance on screen.
R2.2	ATM shows the balance on report.	*added*	R2.2	
R3	ATM withdraws the cash.	*matched*	R3	ATM withdraws the cash.
R3.1	ATM withdraws the cash with limit.	*matched*	R3.1	ATM withdraws the cash with limit.
R4	ATM deposits the cash.	*matched*	R4	ATM deposits the cash.
R5	ATM transfers the money.	*matched*	R5	ATM transfers the money.
R6	ATM quits the transaction	*matched*	R6	ATM quits the transaction.
R7		*matched*	R7	
R8	ATM pays for the services.	*added*	R8	

Figure 6.10: EvoSPL process: evolution activity: requirements specification of NewProduct and the SMD with the change terms of each atomic requirement.

Refactoring point 2.2 of the current FM This refactoring point adds to the next version of the current FM the 'pay_service' feature from the FL of ProductNew, using Refactoring 12. Fig. 6.15 shows the current FM in refactoring point 2.2 that is refined using the feature model refactoring scenario, in the evolution activity. Red asterisk are placed next to the refactoring points, to indicate alterations in the current FM. Relative to Refactoring point 1.1, the current FM in those refactoring points has many changes to the existing SPL based on the changes of ProductNew.

The reverse engineering phase and forward engineering phase of EvoSPL approach derived successfully the current FM of the ATM product family. This model represents the ATM resulting SPL. The mapping phase will be used to relate each feature of the current FM to its implementation code fragments of the code.

ProductNew **VDNew: atomic requirements**	
id	**statement**
R1 'common'	Performs ATM transaction.
R2 'common'	ATM checks the balance.
R2.1 'common'	ATM shows the balance on screen.
R2.2 'optional'	ATM shows the balance on report.
R3 'common'	ATM withdraws the cash.
R3.1 'common'	ATM withdraws the cash with limit.
R4 'common'	ATM deposits the cash.
R5 'common'	ATM transfers the money.
R6 'common'	ATM quits the transaction.
R7 'common'	
R7 'optional'	ATM pays for services.

Figure 6.11: EvoSPL process: evolution activity: variability document of ProductNew with the variability-pattern of each atomic requirement.

ProductNew **VDNew: atomic requirements**	
id	**statement**
R1 'common'	Performs ATM transaction.
R2 'common'	ATM checks the balance.
R2.1 'common'	ATM shows the balance on screen.
R2.2 'optional'	ATM shows the balance on report.
R3 'common'	ATM withdraws the cash.
R3.1 'common'	ATM withdraws the cash with limit.
R4 'common'	ATM deposits the cash.
R5 'common'	ATM transfers the money.
R6 'common'	ATM quits the transaction.
R7 'common'	
R8 'optional'	ATM pays for services.

Figure 6.12: EvoSPL process: applying the visualization technique on the variability document of ProductNew.

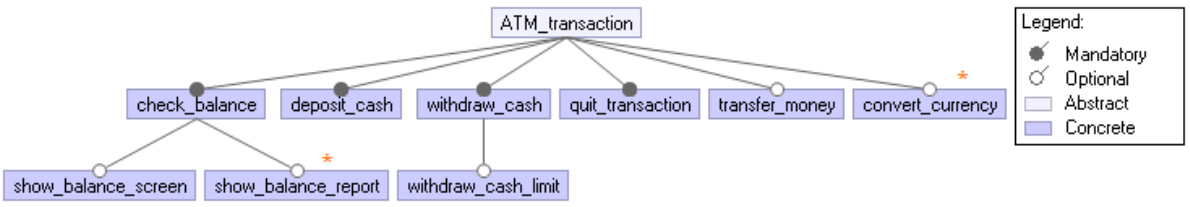


Figure 6.13: The current FM of the ATM products family in refactoring point 2.1.

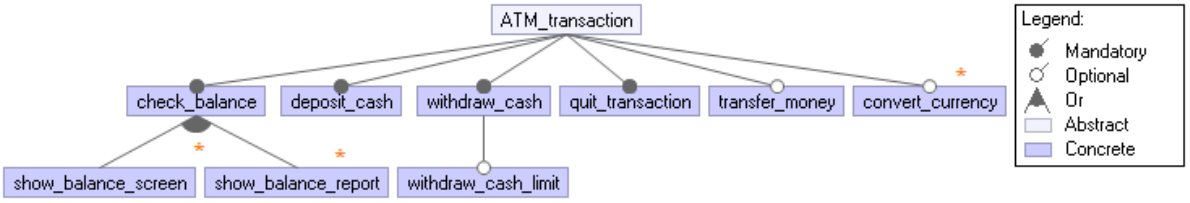


Figure 6.14: The current FM of the ATM products family in refactoring point 2.1.1.

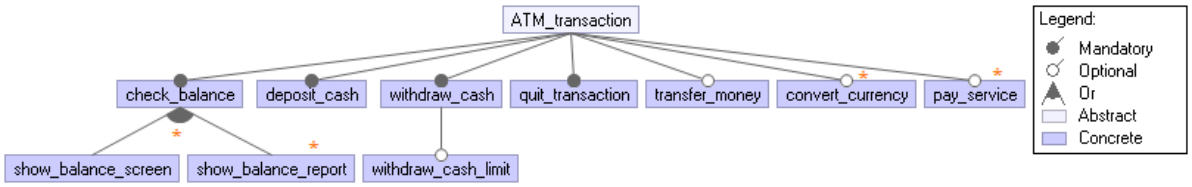


Figure 6.15: The current FM of the ATM products family in refactoring point 2.2.

Chapter 7

EvoSPL: Mapping Phase

This chapter describes the final phase of the EvoSPL approach. It presents the mapping phase that has an important role in feature mapping. It focuses on mapping the artifact coming from the previous EvoSPL phases, the current FM, to the other artifacts of the resulting SPL, the reference architecture and code. This mapping helps to synchronize changes of the current FM with the implementation code, to avoid inconsistencies between them. Thus, this activity relates each feature of the current FM to their locations in the implementation code, using the reference architecture as an intermediate artifact. As shown earlier in Fig. 4.1, the mapping phase follows the forward engineering phase and offers an explicit mapping from the problem space to the solution space of the resulting SPL. The mapping phase is supported by a tool, called friendly Mapper. The tool uses a particular artifact (i.e., traceability tree) as a solution for mapping features belonging to the current FM to code fragments belonging to the implementation code. As shown in Fig. 7.1, this phase uses only one activity to implement the mapping process.

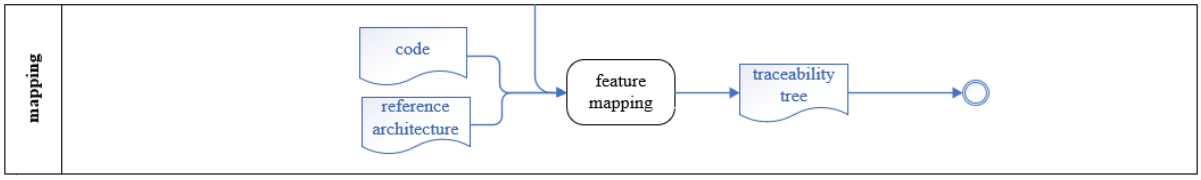


Figure 7.1: EvoSPL process: mapping phase.

7.1 Feature mapping activity

This section describes the feature mapping activity dedicated to the EvoSPL process, to support the feature mapping of the resulting SPL. The activity assumes that features are derived and known upfront, using the previous phases of the EvoSPL approach. As shown in Fig. 7.1, this activity consumes the current FM of the resulting SPL. In addition to the current FM, it consumes the reference architecture, which will be explained in the reference architecture title (see Reference architecture below). It also consumes the code. The feature mapping activity

delivers the traceability tree artifact that not only defines the traceability links (see Traceability links below) between each feature and its code fragments, but also it updates the traces whenever a feature change occurs in the current FM, to preserve consistencies among them.

Activity in brief Having the resulting SPL that owns the reference architecture and the current FM, as well as each product of the family has its implementation code that is cloned from the initial release product and then customized according to customer needs. The mapping between the current FM and the reference architecture is established first, as a centric point, to manage the relationship between the current FM and code, in addition to keep the current FM and code conformance whenever feature changes occur (for more information, see The problem below). As shown in Fig. 7.2, the feature mapping activity (i) takes as an input (see The input and output below) the current FM, reference architecture, and code artifacts, (ii) uses features of the current FM, (iii) traces each feature to its parts of the most upper layer of the reference architecture (see Role of the reference architecture below) and finally (iv) traces the feature from the identified part of the reference architecture to it is corresponding units of the code (see Reference architecture and the code below).

In this way (i.e., the explanation of the previous paragraph), only that units of the identified parts (instead of the entire code base) are analysed to find the code fragments that are responsible for the feature implementation. This leads to (vi) identify the code fragments that implement the feature and store them in the traceability tree (see Traceability tree below). This activity consists of three main macro steps of which the last two macro steps are automated: feature- architecture mapping macro step, feature-code mapping macro step, and feature mapping conformance macro step. However, these macro steps require predefined criteria to be applied successfully (see key criteria to apply the feature mapping below).

The feature mapping activity of the Evo SPL approach applies its macro steps on the most upper layer of the reference architecture, according to the general architecture depicted in Fig. 7.3, it works specifically on Layer 3. Simply, this layer contains the common and variable parts that distinguish each product from the other. For instance, the companies of the automotive domain, usually, customize and configure the modules of the implementation code that are corresponding to the components of the most upper layer (e.g., application layer) where needed, to develop a new product of the family.

Variability and the reference architecture Commonality and variability is captured, at requirements-level, during the reverse engineering phase of the EvoSPL process, which leads to identify the common and variable features (of the resulting SPL) inform of the current FM. The reference architecture includes the variability (variations and variants in form of subsystems

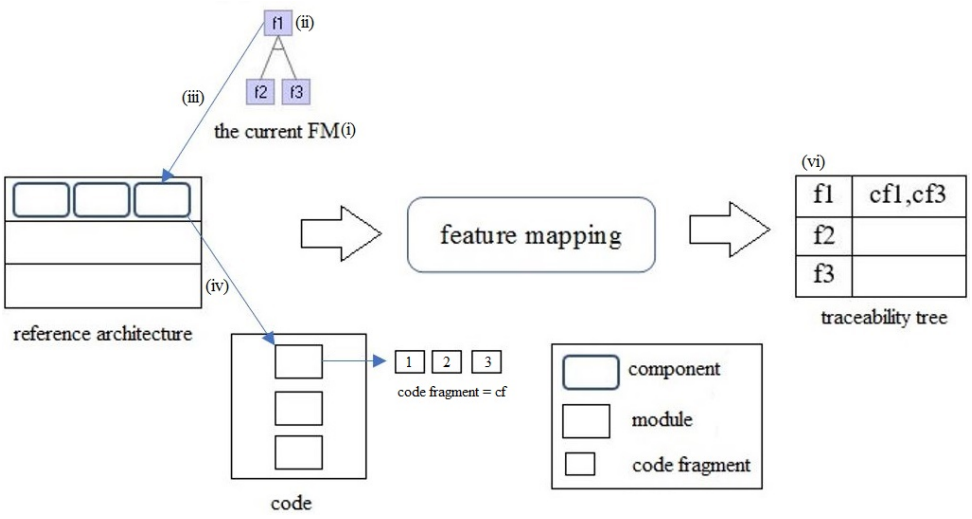


Figure 7.2: EvoSPL process: feature mapping activity.

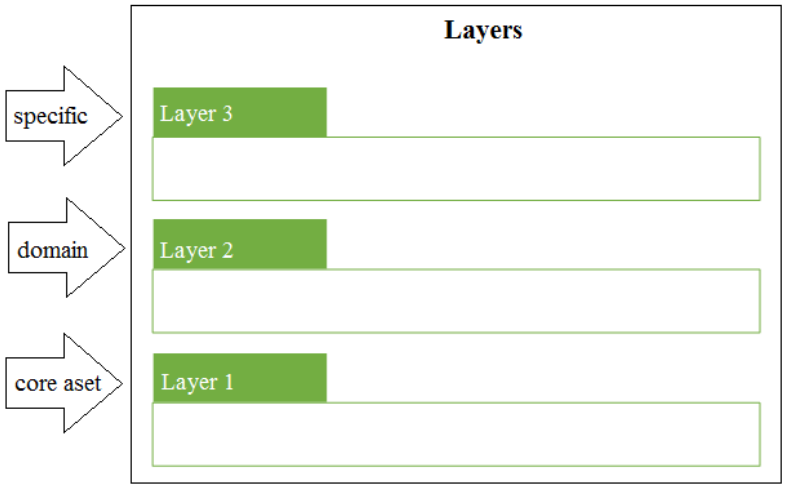


Figure 7.3: EvoSPL process: general structure of the reference architecture of the resulting SPL in the layered architecture pattern.

and components) documented in the current FM. It provides the architectural design of separate products architectures.

Reference architecture The reference architecture includes in its structure the commonality and variability of the resulting SPL documented in the current FM. The EvoSPL approach dedicates its process to product families that have the reference architecture in forms of a layered architecture pattern (see Layered architecture below). It is worth considering this pattern because of its desirable properties for SPLs [Gom11].

Fig. 7.3 shows the general structure of the reference architecture of a products family, which is at the same time the architecture of the resulting SPL that has been extracted from the products of

this family, using the process of the EvoSPL approach. The development view is used to model the architecture structure in layers, subsystems, and components (see Subsystem and component below). It is the main means to decompose the system according to the development view, incorporating the high-level decomposition of the software system into subsystems, components and their relationships (see Fig. 7.4). Hence, this view is the most important one to capture commonality and variability.

Consequently, the variable parts (subsystems and components) of the reference architecture closely correlate with the elements of the current FM (see Fig. 7.5). Typically, the subsystems and components in the (most) upper layer deal with external variability (the variability related to customer and the product requirements). They are only present when the customer needs them. External variability (we called it in this manuscript as variability) is the target variability of the EvoSPL approach. External variability directly contributes to customer satisfaction as customers are aware of this kind of variability and can select those options that serve their needs best. Thus, different stakeholder needs are a cause of external variability. Based on this we are going to continue the explanation of the remaining proposed feature mapping steps.

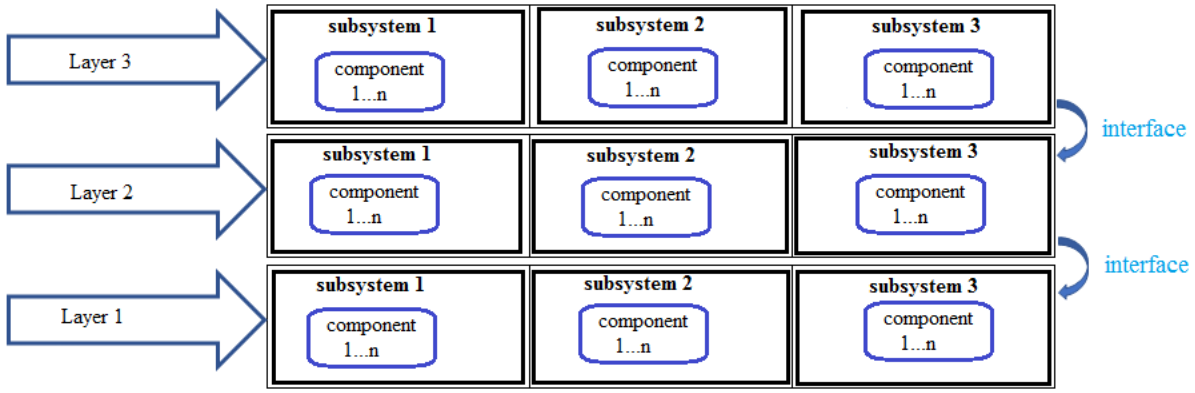


Figure 7.4: EvoSPL process: general representation of layers and subsystems of the reference architecture of the resulting SPL.

We assume in this work that the architect provides restrictions on configurations that are allowed and those that are not, which influences the possible product configurations of the resulting SPL. For instance, the architect uses lists to denote which components are common and which are variable or optional. Moreover, the design structure of the reference architecture normally has several notations to present commonality and variability in the architecture. For instance, a solid line depicts the common part of the architecture. Optional and variant parts are depicted with dotted lines. Another important tool for restricting the number of configurations is the use

of components design. Detailed design is part of development, which must obey the architecture, incorporating the code.

Layered architecture In a layered architecture, normally, each layer has an internal structure consisting of subsystems and components [PBDL05]. Within each layer the variability is determined by the variation in the functionality provided, and by the variation in the functionality provided by the layer below it. In order to design for reusability, the architect typically determines subsystems in such a way that the required variability is encapsulated within their internal components design. This assignment is fixed for all products of the SPL (i.e., the resulting SPL). The structure depicted in Fig. 7.3 includes three layers, namely Layer 1, Layer 2 and Layer 3, each of which has an internal structure consisting of subsystems and components (see Fig. 7.4). Each layer has a specific role and responsibility within the resulting SPL. The layers can be characterized as follows. Layer 1 is the lowest software layer. Its main task is to make the higher software layers independent. It supports a core set functionality (i.e. platform software) of the entire SPL (the common platform for developing architecture). The common platform consists of a set of subsystems and components that exist in the entire SPL products and they enable the creation of new products from the reusable parts (common subsystems and components). This will be the basis for Layer 2. Up to this level all the architecture parts should be common sense and thus are specified.

Layer 2 represents the second level of abstraction, and it provides the subsystems that include components for a basic domain specific functionality (i.e., basic software). Based on this knowledge every company can customize their own company-specific architectural model in Layer 2. Regarding Layer 3, it is the most upper layer of the reference architecture. It allows for various combinations of additional specific functionality (i.e., customer specific software) or sometimes the same functionality in a different context for a product or a set of products. It provides subsystems where each of them is assigned to a functionality that is supposed to be matching the requirements specification and implementation code of the involved functionality, for that it includes components for a specific functionality. In general, Layer 3 of the layered architecture and especially for SPLs contains the product specific part. Normally this layer has a specific part that provides all product specific configurations to other layers, where necessary and provides project specific extensions, if requested by the product and if supported by the platform. In other words, it allows us to determine the specific requirement of a product within the products family, to satisfy a specific customer's needs [Bro+09].

A layered software architecture allows the architect to create independent subsystems and components that address a specific part of the functionality of the whole family. The components communicate through well-defined interfaces that allow for high-performing implementations. This means that all the components are interconnected but do not depend on each other.

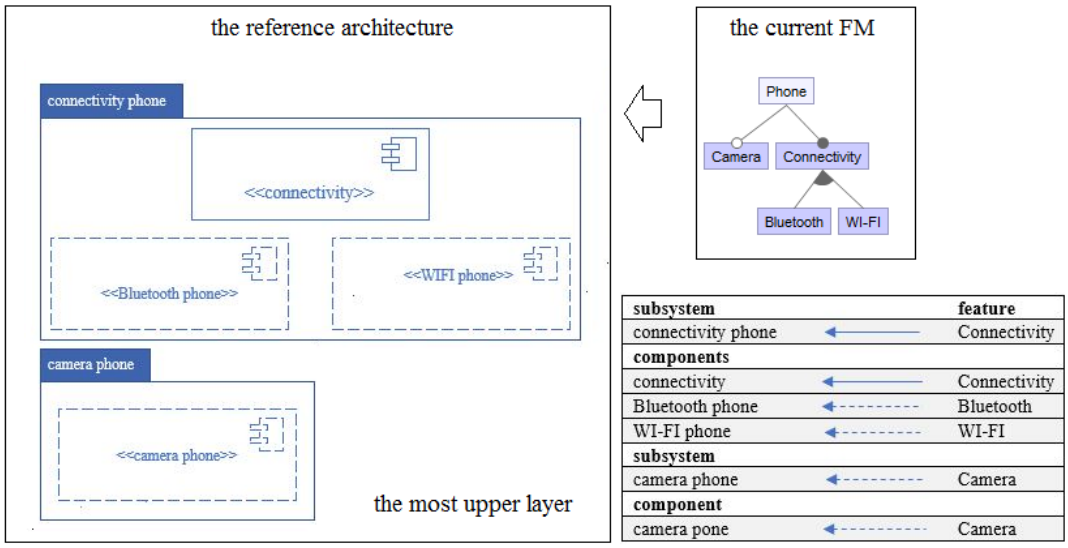


Figure 7.5: The main structure of the mobile phone SPL and their relationship with the current FM.

Subsystems and components Layers and subsystems enable the architect to group similar components. A subsystem is decomposed into a collection of interacting components. In particular, a large part of architecture variability is captured in the subsystems and the components design. The subsystem design denotes the high-level decomposition of the software system into subsystems and their relationships. The structure itself is valid for the entire SPL [PBDL05].

As shown in Fig. 7.4, the subsystems and components within a layered architecture are organized into horizontal layers, each layer performing a specific role within the software system. Although the layered architecture pattern does not specify the number and types of layers that must exist in the pattern, most layered architectures consist of four standard layers: presentation (application), business, persistence, and database. Thus, smaller applications may have only three layers, whereas larger and more complex ones may contain five or more layers.

As depicted in Fig. 7.4, the architecture structure shows a general presentation of layers and subsystems of the resulting SPL. The layers: Layer 1, Layer2, and Layer 3 are present in each product of the resulting SPL. Variability becomes clear in the subsystems and their internal structure (components). The subsystems in Layer 3 deal with variability. They are only present when the customer requires them. So, for example, a product without any functionality related to subsystem 3 has no such subsystem.

In the layered architecture depicted in Fig. 7.4, each subsystem is assigned to a layer matching the abstraction level of the involved functionality. This assignment is fixed for all products of

the resulting SPL. In order to identify variability, software engineers typically can determine the subsystems that encapsulate the required variability within their internal components design. Consequently, the variable parts of the subsystems design closely agree with the elements of the current FM. Typically, the subsystems in the most upper layer (layer 3) deal with variability. They are only present when the customer needs them. Subsystem is decomposed into a collection of interacting components.

The component design provides an internal structure that describes the configurations of components. Fig. 7.4 illustrates the documentation of variability by the subsystems and component design of the most upper layer. Each component may realise a common or optional functionality (feature), or it may realise a variant and is thus only present in a product that provides the functionality for the corresponding variant.

The EvoSPL approach focuses on the documentation of variability in subsystems of the most upper layer (Layer 3) of the reference architecture. The internal structure of this layer consists of subsystems and components. The subsystems in the most upper layer deal with variability that is only present when customers request them. The internal structure of each subsystem may contain common components for the common functionality (common feature) but also there are special components only for variations (optional and grouped features). Fig. 7.5 exemplifies the most upper layer of the 'mobile phone' SPL (and the corresponding current FM) with some services from two subsystems: 'connectivity phone' and 'camera phone'. The 'connectivity phone' subsystem comprises 'connectivity', ' Bluetooth phone' and 'WI-FI phone' components.

The 'camera phone' subsystem has the 'camera phone' component. The 'connectivity phone' subsystem has the common component 'connectivity' for the common 'Connectivity' feature and special components only for the variants. The 'Connectivity' feature describes the connectivity type provided by the 'connectivity phone' subsystem. The variability in this subsystem offers an alternative with two variants that appear in the current FM as 'Bluetooth' and 'WI-FI'. Each of them is realised by a separate component inside the internal design of the 'connectivity phone' subsystem. The former variant is realised by the 'Bluetooth phone' separate component, and the latter is realised by the 'WI-FI phone' separate component. Note that the optional 'Camera' feature is present at one place of the 'camera phone' subsystem in the variable presence of the optional component 'camera phone'.

Notations are used to improve the understandability of the design, as shown in Fig. 7.5, the parts in solid lines compose the common part of the structure. Optional and variant parts are depicted

by dotted lines. The structure determines the generic parts of the resulting SPL, namely the 'connectivity phone' subsystem as well as the optional one the 'camera phone' subsystem. Subsystems are drawn by the use of a package notation (or rectangle) and components are drawn by the use of a component notation (or round rectangle). We are going to use these notations for the entire explanatio

However, this thesis supports an SPL evolution in the automotive domain, for that and normally, the layered architecture with several levels of abstraction fits this concept. The abstraction levels of such architecture allow us to model variability as appropriate, where each level represents a different model of the same products of the family, involving a unique set of components and compositions that are applicable only to a view that abstracts from certain details of implementation.

Finally, another strong reason makes it worth considering a layered (software) architecture is the fact that it describes the software architecture of AUTomotive Open System ARchitecture (AUTOSAR). AUTOSAR is a global development partnership of automotive interested parties [Wik]. It supports an open and standardized software architecture for the automotive domain. Thus, the EvoSPL approach applies its process on product families in the automotive domain, where normally, those families adopt AUTOSAR. Besides, many companies in the automotive domain, such as BMW, Bosch, Ford, General Motors, and Toyota agreed on using it, which makes our approach general and more applicable.

Fortunately, the EvoSPL approach has been deployed at Bosch Car Multimedia company. The case study performed in the company, putting our approach into practice to develop the resulting SPL from an automotive family that has AUTOSAR architecture adopted by the company for a long time in producing its products to automotive consumers. The evaluation section (section 7.3.3) presented in this chapter includes a detailed explanation for AUTOSAR.

Traceability links Traceability links represent the relationships between each feature belonging to the current FM and code fragments belonging to the code of the resulting SPL (see Feature-related-code fragments below). The relationships are established between each feature and the code fragments that are responsible for its implementation in the code (feature-related-code fragments). A traceability link has one-to-many relationships; hence each feature may be implemented by one code fragment or more. In this regard, the feature mapping activity creates a particular artifact to store the relationships, called the traceability tree (see Traceability tree below), and it defines the rules that govern the creations of these relationships. These rules are called tracing rules (see Tracing rules below).

Feature-related-code fragment The traceability tree relates features (of the current FM) and code fragments (of the code), using traceability links. Each link relates a feature to a set of code

fragments that implement this feature. Each feature-related-code fragment is defined, in this work, as a code portion that maintains feature implementation. A code portion is a software unit, such as routine, statement, and expression.

The problem During the evolution of the resulting SPL, the current FM and code can easily become inconsistent in terms of variability [CHGZ+12] [CH+14] [Nev+11] [Gal+11] . For example, the code may keep feature-related-code fragments that are related to a feature previously removed from the current FM. Besides, these artifacts are typically developed and evolved independently during the evolution of the resulting SPL in a way that breaks their conformance with the reference architecture. This problem may occur primarily in the forward engineering phase of the EvoSPL approach, whenever feature changes occur, while there are no mapping links among the artifacts of the resulting SPL, like the current FM, reference architecture, and code. Normally, whenever a feature change occurs (e.g., add a new requirement or feature) a software developer often manually inspects the code to identify the related changes and update them accordingly.

Moreover, a feature represents user-visible characteristics that distinguish the products in the problem space, and a set of feature-related-code fragments implements features in the solution space. To preserve both spaces consistent with each other, whenever changes occur in the resulting SPL, it is worth to map them to each other.

The input and output The feature mapping activity requires the following inputs, to apply its macro steps: (1) features of the current FM (during automation of the mapping phase process, features are taken from the XML file related to the feature tree, which represents the current FM), (2) the development view structure of the reference architecture that contain the variability information (subsystems and components), and (3) the code of the resulting SPL. This activity processes the inputs and produces (macro step 1) the traceability links that map each feature to its design place (subsystem or component) in the reference architecture, (macro step 2) the traceability links that map subsystems and components of the reference architecture to their corresponding packages and modules of the code, and finally (macro step 2) the traceability tree that stores the relationships of each feature to its code fragments of the code.

Besides, (macro step 3) the traceability tree reflects the updates of the relationships whenever change occurs in the current FM. Then based on the traceability tree, the developer can use a proper approach to update the code. This is not the focus of our work, since the research question is how to map the current FM to the reference architecture and code.

Role of the reference architecture In this step of our work, we have the resulting SPL that has been extracted and evolved from the products of a family, which has several product variants and

a new one (or more). Normally, each of them has a specific number of modules that implement the product in the code-level. On the one hand, some modules are common (shared) within the products of a family and those modules comprise the common base of the resulting SPL. On the other hand, some modules are variable (different) within the products of a family, and they are the source of variety for the resulting SPL. As described earlier, the reference architecture in our work has a layered architecture pattern (i.e., consists of layers). Each layer contains a set of parts (subsystems and components), where the common parts are shared within the products of a family and the variable ones are making a product different from another one in the family and the resulting SPL.

In practice, a set of common parts of the reference architecture are usually the base of a products family, which comprises the basis for the resulting SPL design and implementation. Normally, and in our work, the common parts of the reference architecture and their related parts in the code, are corresponding to common features, where the variable parts are corresponding to optional features of the current FM. The role of the reference architecture is to describe the commonality and variability parts among the products of a family and, as such, to provide a common overall structure of the resulting SPL. In custom, the most upper layer of the reference architecture contains the variable parts. They present specific customer and project requirements, which determine associated parts in the code that should be considered in terms of related features. As shown in Fig. 7.6, the feature mapping activity assumes that each subsystem and its integral components of the reference architecture are often implemented as a sub-package and its internal modules of the implementation code. In this work, we mainly focus on variable parts that clearly show the variability of the resulting SPL.

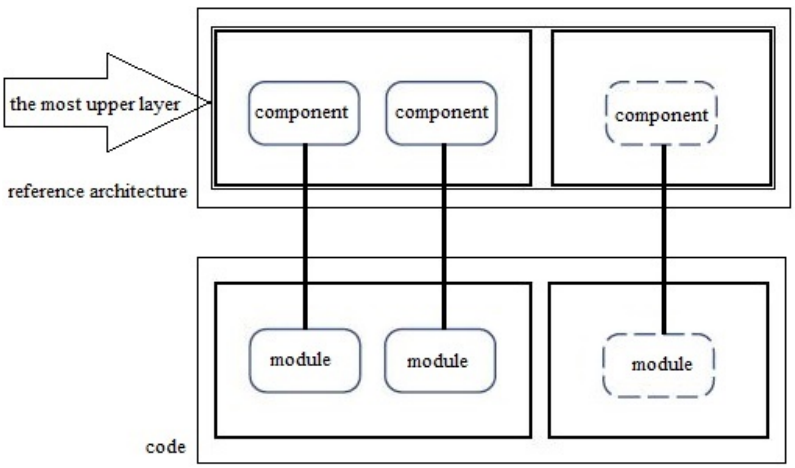


Figure 7.6: EvoSPL process: mapping between the reference architecture and code (component to module).

As shown in Fig. 7.6, the round rectangles and connections drawn with solid lines represent core (common) components (and modules) of the resulting SPL. In contrast, the ones drawn with dashed lines represent variable components (and modules) of the resulting SPL (optional features). Variable components (provide all project-specific configurations to other components, where necessary) and their related modules (which mainly in our case includes C-C++ files) represent the area that the feature mapping activity uses to investigate (find) features positions in the implementation code. Usually, a reference architecture also includes the definition of relations among components (interfaces, dependency, hierarchy, etc.). These relations are not considered in our approach, even though they may contain variability information, but the target of our approach is the feature mapping to the code, hence, the approach only uses the reference architecture as a centric point.

Reference architecture and the code The reference architecture defines the architectural structure for a set of related products (of the resulting SPL) [Gar+03]. As explained before, the reference architecture contains common parts (subsystems and components) and their corresponding parts (packages and modules) of the code shared within a products family/the resulting SPL. Also, they comprise the common base (a set of core functionality) of the resulting SPL. At the same time, the reference architecture and code contain variable parts and their corresponding parts of the code. The variable parts make each product different from the other. The common and variable parts in both artifacts (the reference architecture and code) can be defined in terms of variability model realized in our case by the current FM [Gom13].

key criteria to apply the feature mapping Generally, the usage of feature mapping activity macro steps is possible due to the reference architecture structure. In this title, we present the criteria to check in the working environment, regarding the resulting SPL, prior applying the feature mapping.

Criteria 1. The reference architecture of the resulting SPL should have a decomposition style that forms a layered architecture pattern. Such a layer should provide an interface offering services between the layers. Layers have strict ordering such that a higher layer is only allowed to use the next lower layer.

Criteria 2. The reference architecture is typically modelled as subsystems and components. Some of them are presented in all the products (commonality) and the others are variable (bound to customer and product requirements). Components can be connected through interfaces that restricts the number of component configurations.

Criteria 3. A functionality (feature) is typically assigned to a subsystem or component of the reference architecture that encapsulates the implementation of functionality from the automotive domain. This assignment is fixed on the resulting SPL.

Criteria 4. Each product of the resulting SPL accompanies with a code that follows the project hierarchical level structure explained in the Implementation code elements title.

7.1.1 Feature- architecture mapping macro step

The feature-architecture mapping macro step presents a solution to map features of the current FM to the reference architecture. This solution aims to define the concept of traceability between the current FM and reference architecture. This macro step identifies the feature-related-arch-elements (arch is the abbreviation for architecture), namely the feature-related-subsystems and feature-related-components of the reference architecture. This macro step also uses the traceability links to document the relation between features and the feature-related-arch-elements. The links enable, for instance later, to search only the feature-related-modules of the code that are corresponding to the feature-related-components of the reference architecture (instead of the entire code base), to find the code fragments that are responsible for the feature implantation.

Practically, our approach considers that feature mapping to the reference architecture is straightforward, based on the following reasons. First, the structure of the reference architecture is predefined and agreed by the team developer (or company) of a product family. Second, the architect designed the reference architecture to fulfil almost all the requirements given for a products family. Finally, detailed design of the reference architecture is part of development and produces the code, which almost obeys the architecture, incorporating the code view. The developer can identify from the design of the reference architecture the components that encapsulate the common and specific functionality for the products of a family. Thus, the feature-architecture mapping macro step follows the structure of the reference architecture as a guidance to find the intended feature-related-arch-elements using the following steps.

Step 1.1 provides a traceability link between features of the current FM and the layer of the reference architecture that contains variability and customer (project) specific requirements. As shown in Fig. 7.7, normally, the most upper layer (layer 3) is the target and proper layer to perform the feature mapping.

Step 1.2 provides traceability links between each feature of the current FM and the feature-related subsystem of the most upper layer. This step retrieves a set of the ARs that specify the feature and uses the statement parts of them to determine the feature-related subsystem (see Feature-related-design below).

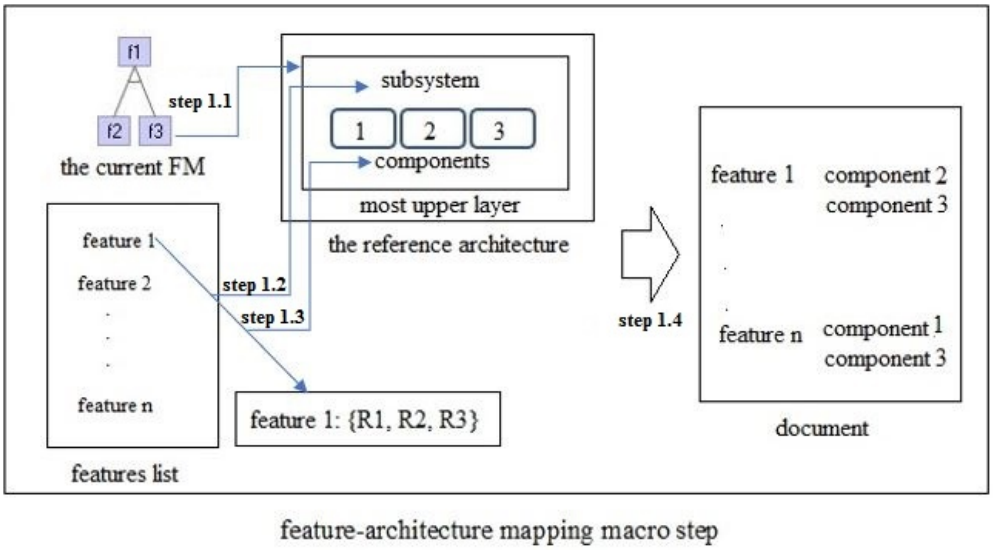


Figure 7.7: EvoSPL process: feature-architecture mapping macro step.

Step 1.3 provides traceability links between each feature of the current FM and the feature-related component in the subsystem that is already identified for the feature, when required. This step uses the retrieved ARs that specify the feature and uses them, but this time, to determine the feature-related component (see Feature-related-design below).

Step 1.4 stores the traceability links of each feature to its feature-related-arch-elements in a document.

As shown in Fig. 7.7, this macro step consumes the features of the current FM and the reference architecture and delivers a document that contains the list of features of the current FM and their feature-related-arch-elements. This document will be used as an input to the feature-code macro step.

Feature-related-design Basically, a large part of the commonality and variability in the reference architecture originates mainly from the commonality and variability in requirements. The architect determines how requirements, including variability, are reflected in the reference architecture. Thus, step 1.2 and step 1.3 (1) define and maintain the traceability definition in a backward direction from a feature to the ARs that specify this feature, by taking into account the statement of each AR (see Feature-requirements mapping below). Then step 1.2 and step 1.3 (2) uses this traceability to realize feature-related-arch-elements that document the feature while ensuring that they satisfy the product requirements.

Feature-requirements mapping Defining the ARs of each feature involves a backward mapping from features of the current FM to ARs of the FL. Fig. 7.8 illustrates the basic steps for

performing the backward mapping from a features to its related ARs, in order to finally map the feature to feature-related-arch-elements. The five steps are as follows.

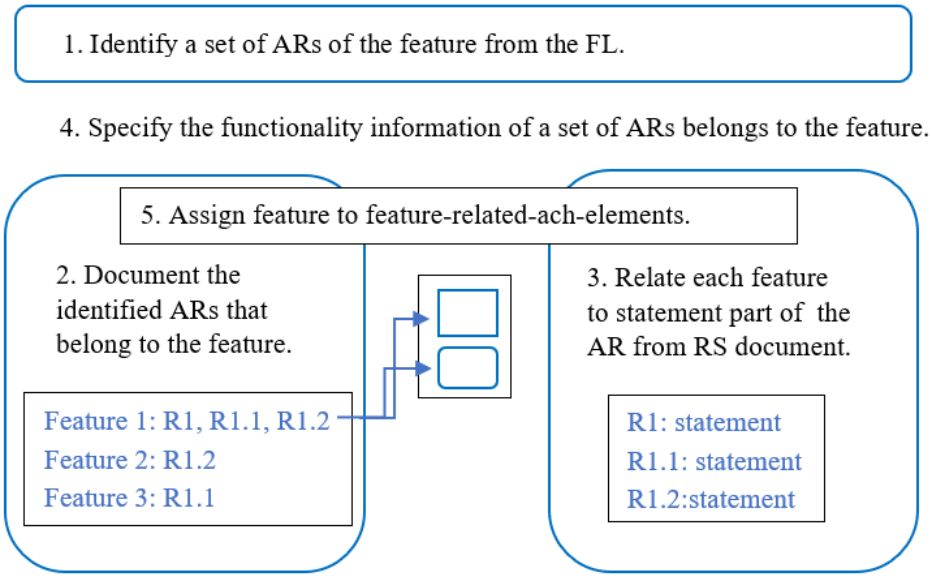


Figure 7.8: EvoSPL process: feature-requirements and feature-related-arch-elements mapping.

1. Identify a set of ARs of the feature from the FL.
2. Document the identified AR that belongs to the feature.
3. Relate each AR to its corresponding statement part from the RS document.
4. Specify the functionality information of a set of ARs belonging to the feature.
5. Assign feature to feature-related-arch-elements.

Steps 2 and 3 are repeated until all the ARs have been considered.

A set of ARs may specify a feature of the current FM. Each AR (see Atomic requirement structure of the reverse engineering chapter) has a statement part that (partially) shares to specify a specific functionality of the product. A set of ARs specify one functionality (feature) of a product of the resulting SPL, which has a corresponding design in the reference architecture that models this functionality. Thus, step 1.2 can map a feature to its related feature-related-arch-elements (subsystems and components) that encapsulates its functionality, based on (1) feature-requirements mapping, (2) architect of the reference architecture can give a useful information and consultation regarding this issue, and (3) the name of a feature can be used to support such mapping. According to the guidelines of our approach, the feature has a meaningful name that reflects a clear property or functionality of the resulting SPL.

7.1.2 Feature-code mapping macro step

The feature-code mapping macro step aims to satisfy the main goal of the feature mapping activity. It maps features of the current FM to the code of the resulting SPL, but not the reverse. It establishes a kind of traceability link between each feature and its implementation code fragments, using the reference architecture structure design, which has been investigated in the feature-architecture mapping macro step, as a guidance. Mainly, this macro step uses the output document of the feature-architecture mapping macro step to locate each feature of the current FM to its implementation positions in the code. The feature-code mapping macro step consists of the following steps.

Step 1.5 investigates the code characteristics and code variability nature of the products of the resulting SPL.

The code characteristics The feature mapping activity assumes that one way to achieve a successful mapping between the reference architecture and code is to ensure that the code reflects the abstract concepts shown on the reference architecture design, which can be achieved by supporting subsystems and components design in its layers. Thus, the Evo SPL approach deals with the code of the resulting SPL that has the following characteristics.

- The code of each product is organized by a main package, which marks the product name, as well as composed of other packages, where each of them corresponds to a layer of the reference architecture.
- The package corresponding to the most upper layer of the reference architecture has sub-packages (sub-package is a package within the main package). Each sub-package corresponds to a subsystem of the reference architecture. Normally, this package has a unique and common name among all the products, as well as it has a name identifier close to the name of the most upper layer of the reference architecture.
- Each sub-package contains one or more modules (a piece of software that has a specific functionality), where each module has a specific functionality of the product and corresponds to a specific component of the reference architecture.
- Each module contains one or more different source files shared to implement the functionality of the module and can be edited separately.
- Each source file has a name that reflects its functionality and reflects the module and sub-package that the source file belongs to. The source files correspond to internal design of the component. They are the target place of the feature-code mapping macro step, to search for the feature implementation.

- The implementation code elements (see Implementation code elements below) follow the programming naming convention, which is a set of rules for choosing the character sequence to be used for identifiers that denote package, source file, routines, and other entities (local and global variables) in code and documentation. By convention a company, especially in the automotive domain, uses its reserved domain names for its package, source file, routine, and attribute names, which closely correlates with the reference architecture.

Fig. 7.9 exemplifies an example that illustrates the implementation code characteristics and Fig. 7.10 exemplifies, with the same example, how the code view correlates with the reference architecture. The main package (phoneProduct 1) hosts the entire code implementation of the product and has the product name. Each sub-package (e.g., phoneConevtivity) encapsulates the source files (e.i., connect1Blutooth.cpp and connect2Blutooth.cpp) members related to a specific module (i.e., connectBlutooth), which allows us to divide the product into discrete units of functionality. A product can have one or many modules and one module may use another module as a dependency. Each module can be independently built, tested, and debugged. In our case, the code view (see Code view below) obeys the development structure of the reference architecture and has a relation with its elements.

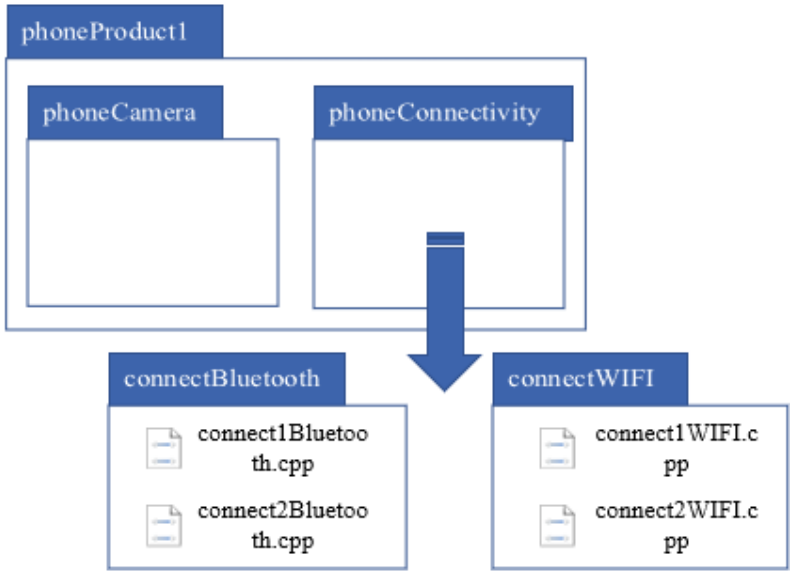


Figure 7.9: An example that illustrates the implementation code characteristics.

For example, as shown in the lower side of Fig. 7.10, the 'phoneConnectivity' sub-package of the 'phoneProduct1' main package corresponds to the 'connectivity phone' subsystem of the reference architecture in the upper side of the same Figure. The sub-package and its corresponding

subsystem follow the conventional name standards that give names reflect both the functionality and relation between them.

The code view The products of the resulting SPL have a code implementation that follows a file system hierarchy, which obeys the reference architecture, incorporating the code view. The code view presents the code artifacts after development and determines the interrelations between them as follows.

- The resulting SPL consists of products, each product has a platform implementation code that contains common sub-packages (e.g., 'phoneConnectivity' sub-package corresponds to 'connectivity phone' system of Fig. 7.10) that reside in the main package and reflects the resulting SPL core assets. The main package also contains other sub-packages (e.g., 'phoneCamera' sub-package corresponds to 'camera phone system' of Fig. 7.10) that reflects the product specific requirements and configuration.
- The implementation code platform relies on hierarchical file systems to manage the code view that closely correlates with the development view of the reference architecture. The strategy is as follows.
    1. Each layer corresponds to a package that has a name. The name reflects the layer name and contains other sub-packages. It is worth to mention that the feature mapping activity focuses on the most upper layer, in consequence, it focuses on the package that corresponds to this layer, to search for features-related-code fragments.
    2. Each sub-package corresponds to a subsystem of the reference architecture and has a name that reflects the subsystem name and contains other modules.
    3. Each module corresponds to a component of the reference architecture and has a name that reflects the component name and contains source files that implement the functionality of the module.
    4. Each source file consists of routines and other software units (variables, statements and expressions).
    5. A set of routines (function or method) from one source file or more corporates to perform a specific functionality. Normally, routines use the local and global variables as well as expressions and statements to fulfil a complete runnable implementation.

Fig. 7.10 shows in the lower side the code implementation of 'phoneProduct 1' (one product of the 'mobile phone' SPL presented earlier in this chapter). The code implementation follows a file system hierarchy. The code view presented in the lower side of Fig. 7.10 obeys the reference architecture presented in the upper side of the same figure. The product code implementation resides in the 'phoneProduct1' main package and consists of two sub-packages, namely

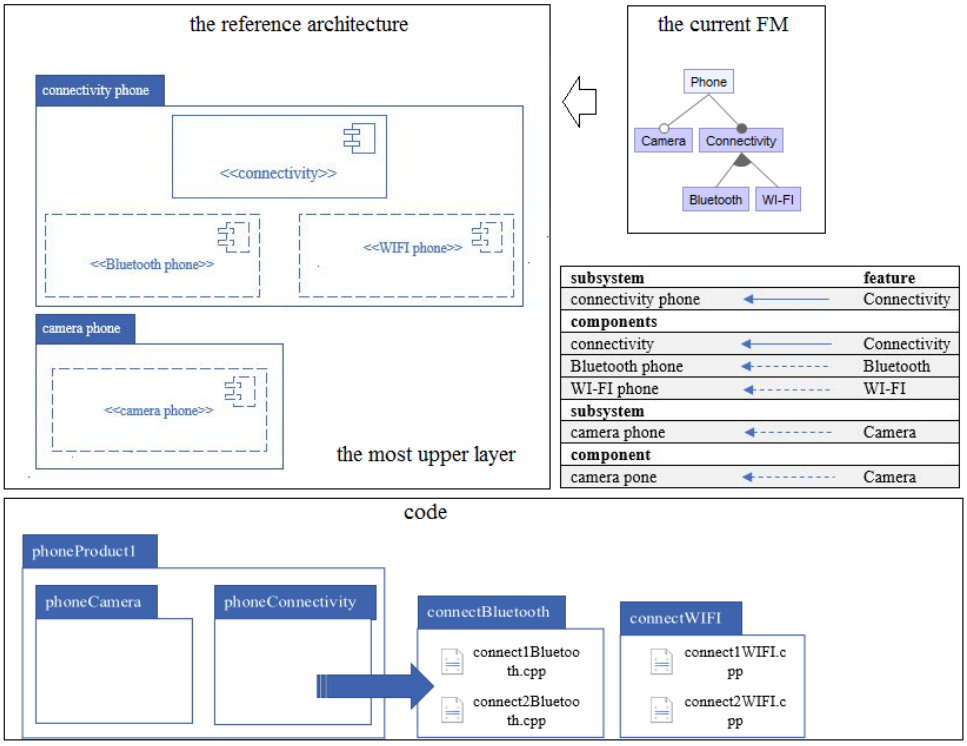


Figure 7.10: The interrelation between code view and the reference architecture.

'phoneConnectivity' and 'phoneCamera'. Each sub-package has a unique name that has two parts. The first part reflects its functionality and the second part reflects its parent package. For example, the 'phoneConnectivity' performs a functionality related to phone connectivity ('connectivity') and its parent package is 'phoneProduct'. The 'phoneConnectivity' sub-package corresponds to the 'connectivity phone' subsystem of the reference architecture. Both have approximately a common name identifier that reflects their relationship.

As shown in the lower side of the figure, each sub-package contains one module. Each of them performs a specific unit of functionality, as well as, each of them corresponds to a specific component in the reference architecture (e.g., the 'connectBlutooth' module corresponds to the 'Bluetooth phone' component), as shown in the upper part. The source files are stored in a module, and they implement its specific functionality. Source files contain the code that is documented by the internal design of the component, which is out the scope of this research work. The 'connect1Bluetooth.cpp' share with the 'connect1Bluetooth.cpp' source file in the 'connectBlutooth' module to implement the module functionality. Based on the above explanation, the 'Connectivity' feature of the current FM (shown in the upper part of Fig. 7.10) has a traceability link with 'connectivity phone' subsystem' of the reference architecture, which can be used as a guidance to trace the feature to the 'phoneConnectivity' sub-package. The 'phoneConnectivity' sub-package is searched to locate the 'Connectivity' feature implementation inside its contents.

The code variability nature As mentioned in the previous chapters, this thesis deals with product families of the automotive domain that have been implemented in C-C++ programming language. In order to find features positions in the code. Step 1.5 reviews and analyses the variability nature of the code. This analysis helps to achieve an understanding on how code works and implements variability (features). The code variability investigation results in the following observations.

1. The pre-processor code supports file inclusion (#include), macros constant (#define), and conditional compilation. In practice, mainly, the code uses #ifdef blocks for variability realization, where code variants are conditionally included or excluded by #ifdef statements.

2. While variations using Conditional Compilation are implemented as #ifdef blocks in the code, the Conditional Execution is used also as if-else blocks. If there are more than two alternatives, then the variation is implemented as a switch block. It is often used for realizing features that are configured after compilation (e.g., using a configuration file).

3. While the code of the source files is coarse-grained with routines, the implementation of routines is fine-grained, because they are implemented as if-else blocks and conditional code blocks. The conditional compilation implements fine-grained variability by inclusion or exclusion of #ifdef-blocks within the code of source files , as well as it includes many individual programming statements (variable declarations and expressions).

4. Many features can be enabled/disabled for the product, in the configuration file of the code that relates to this product. Once the product is created the product-specific configuration or modification must be done using file inclusion. Normally, this inclusion contains a configuration and setting some values for the features. Hence, we are interested in features and its related code fragments, these files are not a valuable source for feature identification.

5. The macro definitions and #ifdef blocks are the most-frequently-used features of the pre-processor. The #ifdef blocks have been used to delimit the optional code associated with each feature, such as #ifdef and #endif to surround the code fragments.

6. The macro constants are defined with directive #define and are normally assigned values. Then the constants are used in #ifdef statements, which allow #ifdef blocks to be compiled conditionally into the corresponding code fragments. Generally, #ifdef block can be mapped to a variation that is either optional or an alternative feature.

7. The code fragments that are responsible for a specific feature implementation may crosscut several files and routines of the code.

Implementation code elements The feature mapping activity assumes that the implementation code relies on hierarchical file systems to manage source files based on project hierarchical level of code elements, which implement features ranging from higher hierarchical implementation level according to (C-C++) programming grammar, such as packages, source files, and routines to lower hierarchical implementation levels, such as statements and expressions. The hierarchical level of code elements that implement features are viewable as the following software units.

- Package/sup-package/module contains entirely all the source files that perform a specific functionality as implementing a set of features.
- Source file contains entirely all the routines and pre-processor directives (that exist in C-C++, including #ifdef, #ifndef, and #else) as implementing a set of features. Basically, these directives indicate to the programmer whether code fragments they delimit should be passed to the compiler or not. In this way, it is possible to generate a product from the resulting SPL with optional features indicated by software developers.
- Routine in source file implements a feature (i.e., signature and body). The routine may combine the implementation of more than one feature. At the same time, a feature implementation may crosscut several routines, which is based mainly on the code nature.
- Attribute implements or share to implement a feature.
- Statement implements or shares to implement a feature, including method calls, assignments, conditional statements, loop statements, etc.
- Expression implements or shares to implement a feature (e.g., the expression of an if or loop statement).

Fig. 7.11 presents the resulting SPL with four products, namely P1, P2, P3 and P4. Using conditional compilation, macro constants are defined with the directive #define and are undefined with #undef directive, as depicted in the left-hand side and right-hand side of the figure. Then the constants are used in an #ifdef statements, as depicted in the middle of the figure.

To implement variability in the resulting SPL, constant definition and undefinition usually occurs in product code (P1, P2, P3 and P4), while constant usage usually occurs in core code, as shown in Fig. 7.11. In this way, when the core code is reused and instantiated by products, different variant elements in #ifdef blocks are conditionally compiled into corresponding product code. According to macro constants definition occurs in each product, when the core code is reused, the resulting SPL will have the following product configurations: P1: {connectivityPhone, Bluetooth}, P2: {connectivityPhone, Bluetooth, camera}, P3 {connectivityPhone , WIFI}, and P4 {connectivityPhone , WIFI, camera}.

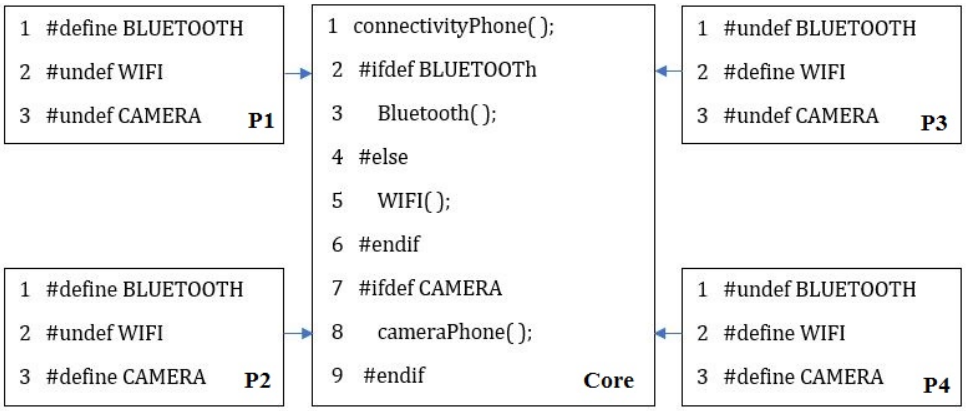


Figure 7.11: An example of variability realization in C code using the conditional compilation.

Step 1.6 creates five tracing rules that define how relationships can be established between features of the current FM and code fragments of the code. In other words, the rules restrict a feature of the current FM that can be traced to which code fragment(s) of the code.

Tracing rules The tracing rules defines how to relate features belonging to the current FM to code fragments belonging to the code. The rulers restrict for the feature how to find the places of its implementation code fragment(s). The tracing rules results in traceability links. Each traceability link relates one feature to its implementation code fragments. The tracing rules defines the restrictions to create traceability links that consider variability. From the current FM side, the variability is specified by means of optional and alternative features. From the code side, the variability is specified by (1) means of relationship between variable components and its corresponding modules of the code and (2) variability realization mechanisms of the code, such as conditional compilation (see The code variability nature). Based on the traceability links that have been established in the feature-architecture mapping macro step and its resulting document, the tracing rules can be defined as follows.

1. Tracing rule (1). All the features of the current FM can trace to a package (i.e., the package corresponds to the most upper layer of the reference architecture) containing the implementation of a specific product requirements inside the main package of a product or more.

2. Tracing rule (2). A feature can trace to the sub-package corresponding to the feature-related-subsystem.

3. Tracing rule (3). A feature can trace to the specific module corresponding to the feature-related component in the subsystem that is already identified for the feature.

4. Tracing rule (4). A feature can trace to a set of source files related to the module that is already identified for the feature.

5. Tracing rule (5). A feature can trace to a set of routines, statements, and expressions related to the source file that is already specified for the feature.

As a side result of the tracing rules, we give the following recommendations for a feature location title presented in step 1.7.

- Normally, each source file of a module implements a set of related features (i.e., parent-feature and its child-feature).
- Normally, each routine of a source file implements a specific feature or a set of related features. Also, a feature implementation may crosscut many routines in the same file or in other files of the same (or another) module.
- Constants, variables, and expressions (which are situated either in the source files or in the routines) may implement a feature. In other words, a feature implementation may be mapped to a single constant, statement and expression or their combinations.
- Normally, common code fragments implement a common feature.
- Naturally, conditional compilation directives implement an optional or alternative feature. They are annotated using #ifdef blocks, depending on its evaluation, the feature-related-code fragments are either included for compilation or not.
- Normally configuration files reveal optional features. Those files contain the constraints and dependencies of features, and they contain directives that run a specific feature that makes each product unique.
- Comparing the code of source files of two products, using a proper compression tool, reveals obviously optional and alternative features. Alternative features cannot be implemented in more than one product.

Step 1.7 investigates and locates the feature positions in the code. This step uses feature location of the EvoSPL, to locate feature- related-code fragments that are responsible for feature implementation.

Feature location Given the products of the resulting SPL that has some complexity as well as the current FM. In this context, knowledge about features of the resulting SPL can be scattered across the code that somehow is organized. If this is the case, step 1.7 performs feature location

and utilizes the following resources, to enrich the feature location process (1) the configuration files, (2) the requirements documents, and (3) the code comments (contain rich documentation that helps to find feature implementation in the code). Software engineers can find feature places (i.e., feature-related-code fragments) through a manual analysis of the code assisted by Eclipse IDE, using the following feature location steps.

1. Use Eclipse to import packages of the products of a family into its local development workspace. Our approach does not require any changes to code of a specific products family.
2. Label each feature of the current FM with its related product (see Labelling the current FM below).
3. Use the current FM, which has been labelled with feature-related-product and the traceability links that are established as a result of the tracing rules of step 1.6, to find the feature positions (feature-related-code fragments) in the code of the feature- related-product.
4. Prepare feature-related-terms, which are a set of terms (target words) related to or have a great significance to feature (see Feature-related-terms below).
5. Search the code of the feature-related- product for the feature-related-terms and retrieve the feature-related-code-fragments. The search process targets the code in the source files of the feature-related-product labelled in the current FM. The search process uses the search capability of the Eclipse, to locate and return feature-related-code fragments. The Eclipse IDE search capability highlights the feature-related-code fragments using code visualization capability of the same IDE (see Eclipse below).
6. Relate the feature with feature-related-code fragments, once a matching appears, by highlighting the places of the code that matched the feature- related-terms.

Labelling the current FM Fig. 7.12 summarizes our objective. On the one hand, it contains the features of the resulting SPL, and on the other hand, it has a label besides each feature. The label specifies the feature-related-product that have (implement) each optional and alternative features. Software engineers can use the FL of each product to fill the labels. Software engineers can follow one of the two strategies listed below when labelling each feature.

1. Software engineers can label a common feature with the initial release product (e.g., using 'i' character).
2. Software engineers can label an optional or alternative feature with one of the products that have (implements) this feature. The EvoSPL assumes that a feature has the same implementation in all the products of the resulting SPL.

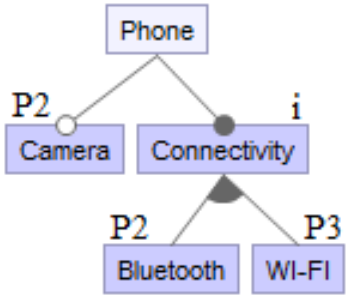


Figure 7.12: The current FM labelled with feature-related-products.

Feature-related-terms Software engineer can prepare and review a set of terms related to each feature from artifacts of the feature-related product, like the manual guide, requirements document as well as RS document - ARs and commenting code may contain meaningful terms. The code comments contain a rich documentation that helps to collect significant feature-related-terms that help to specify the feature-related-code-fragments. For example, a set of predefined feature-related-terms may include the feature name itself, technology, signal, service, capability, and attribute.

Commenting the code Commenting code is a practice that normally uses single-line notes throughout the code. These notes are called comments. A code comment is a piece of human-readable text that is written in the source to explain something about the code. The following is an example of code commenting.

Example 6.1: a code comment
// increment variable comment
variable++;

Step 1.8 creates traceability links from the feature to its positions in the code. Each traceability link represents a relation between a feature and the set of code fragments implementing the feature.

Step 1.9 stores the traceability links in the traceability tree (see Traceability tree below).

Repeats step 1.7 to step 1.9 until all the features of the current FM are considered.

However, it is difficult to ensure that the feature location can find all the feature positions in the code and the feature-code mapping macro step can trace each feature to all the feature-related-code fragments. Multiple passes and search through the code is required to mitigate such difficulties.

Eclipse The EvoSPL approach uses Eclipse, which is an integrated development environment (IDE) used in computer programming [Ecl], to perform the steps of feature-code mapping macro step. Eclipse primary use is for developing Java applications, but it may also be used for developing applications in other programming languages, including C-C++. Following Eclipse IDE terminology, each product is called a package. Thus, we suggest using search capabilities of the Eclipse IDE to locate feature-related-code fragments using the following stages.

Stage 1. The initiation stage uploads product packages to Eclipse workspace. Apart from this stage is to configure the search setting; the contain text; the match case of the search (cases sensitive, regular expression, or whole text); and search scope (e.g., workspace).

Stage 2. The searching stage uses feature-related-terms (target words) to search the packages for feature-related-code fragments. Eclipse search capability lists all the feature-related-code fragments that are containing 'target words' occurrences. During this stage, software developers can take notes related to features and features dependencies (constraints) to verify their conformance with ones that are derived using reverse engineering phase.

Stage 3. The documentation stage stores the relation between feature and feature–related-code fragments in the traceability tree.

Traceability tree Our approach maps the abstraction traceability gap between features and code of the resulting SPL, by relating them in an artifact called traceability tree. This helps software developers to avoid the difficulty of maintaining consistency between the current FM and the code that evolves independently. The traceability tree is a kind of dropdown tree starting its third level at feature. It allows software engineers to trace each feature to the feature-related-code fragments with their positions, just by dropping down each level, starting at SPL project, the resulting SPL package, features, and moving toward code fragments. The traceability tree provides many traceability links. Each link relates a feature to the feature-related-code fragments (see window 1 of Fig. 7.27).

The EvoSPL approach presents the evolution of the resulting SPL as a sequence of the current FM at different points. At each point, the approach refines the current FM with the features of a new product. Thus, another important capability of our approach is maintaining the traceability links in the traceability tree (automatically) when a feature change occurs (e.g., feature added or removed). The new features of the traceability tree are presented with red color, to give software engineers an alert to update the feature-related-code fragments whenever features change occur, which enables the current FM and the code of the resulting SPL to be linked and to be evolved consistently. Thus, we develop an Eclipse-based tool, named friendlyMapper, to support the concept of feature-code traceability links through the traceability tree. This and other potential capabilities of the tool are discussed in section 7.3.

7.1.3 Feature mapping conformance macro step

The feature mapping conformance macro step maintains the current FM and code conformance whenever feature changes occur (e.g., add new feature). This macro step can (automatically) update the traceability links of the traceability tree, to maintain the coherence between features and code fragments, when existing features are changed. The feature mapping conformance macro step works on the already established traceability links of the traceability tree. It works in case of features change occurring in the current FM. Thus, it (Step 1.10) applies the predefined steps for each case as follows.

Case 1 (add feature). Whenever a feature change occurs to the current FM , which involves adding a new feature, then the feature mapping conformance macro step (1) adds this feature to the features level of the traceability tree, (2) gives the new feature a red color, and (3) creates new links from the new feature to feature-related-code fragments.

Case 2 (delete feature). Whenever a feature change occurs to the current FM , which involves deleting a feature, then the feature mapping conformance macro step (1) deletes this feature from the features level of the traceability tree, and (2) removes all the tractability links from this feature to feature-related code fragments of the traceability tree.

An important capability of friendlyMapper tool (section 7.3) is creating and maintaining a feature and feature-code fragments relationship. It supports operations such as creating traceability links, adding traceability links, and removing traceability links. All the changes occurring to the current FM are automatically reflected and saved in the traceability tree. Thus, further explanation for the feature mapping conformance macro step is presented in the tool support section of this chapter.

7.2 An illustrative example of the mapping phase: ATM products family

This section presents the illustrative example of the mapping phase. It provides further clarifications of the EvoSPL approach, based on the previous illustrative example presented in the earlier chapters. This example supports the feature mapping of the ATM resulting SPL, which has been extracted and evolved during the previous phases of the EvoSPL approach. Consider the four products, including the features set of each product (shown in the upper side of Fig. 7.13), the current FM (shown in the lower side of Fig. 7.13), and the reference architecture (shown in Fig. 7.14), this example aims to apply the feature mapping activity and delivers the traceability tree artifact. The traceability tree not only defines the traceability links between each feature of the current FM and feature-code fragments of the code, but also it updates the traces when a feature change occurs in the current FM, to preserve consistency among them.

product name	features of the product
Product 1	{check_balance, withdraw_cash, deposit_cash, quit_transaction}
Product 2	{check_balance, withdraw_cash, deposit_cash, quit_transaction, convert _currency}
Product 3	{check_balance, withdraw_cash, deposit_cash, quit_transaction, show_balance_screen, withdraw_ cash _limit, transfer_money}
ProductNew	{check_balance, withdraw_cash, deposit_cash, quit_transaction, show_balance_screen, withdraw_cash_limit, transfer_money, show_balance_report, pay_service}

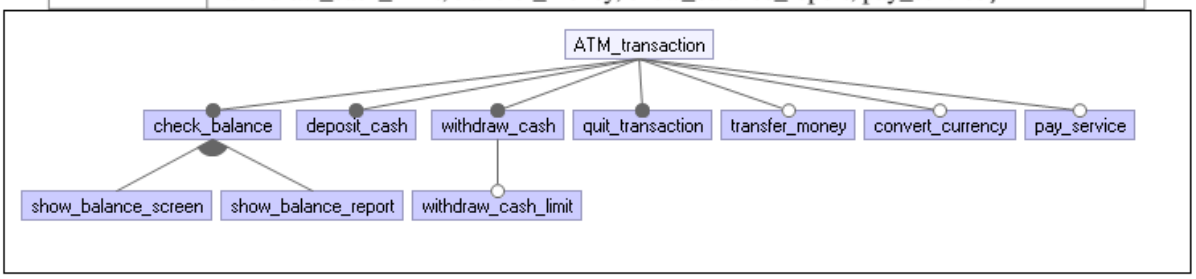


Figure 7.13: Features set of the products and the current FM of the ATM resulting SPL.

The mapping between the current FM of the ATM resulting SPL and the reference architecture is established first, as a centric point. Fig. 7.14 shows the general structure of the reference architecture of the ATM resulting SPL that has been extracted from the products of the same family, using the process of the EvoSPL approach. The development view is used to model the architecture structure in layers, subsystems, and components. The reference architecture composed of three layers: (1) resource management layer, (2) system service layer, and (3) the ATM transaction layer (the most upper layer). The mapping phase focuses on the most upper layer of the reference architecture to perform its unique feature mapping activity, which consists of three macro steps.

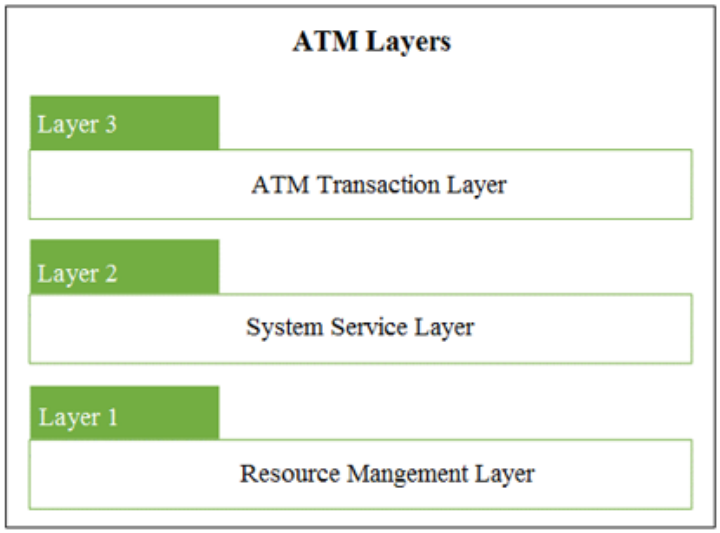


Figure 7.14: General reference architecture of the ATM resulting SPL.

Fig. 7.15 exemplifies the 'ATM Transaction' most upper layer of the reference architecture with some services from four subsystems: 'Balance Transaction', 'Cash Transaction', 'Service Transaction', and 'Transaction Synchronization'. The 'Balance Transaction' subsystem comprises 'Balance Checking', 'Screen' and 'Report' components. The 'Cash Transaction' subsystem has 'Cash Withdrawal', 'Withdrawal Limit', and 'Cash Deposition' components. The 'Service Transaction' subsystem contains 'Money Transfer', 'Currency Conversion' and 'Service Payment' components. Finally, the 'Transaction Synchronization' subsystem owns 'Quit Transaction' and 'Change Transaction' components.

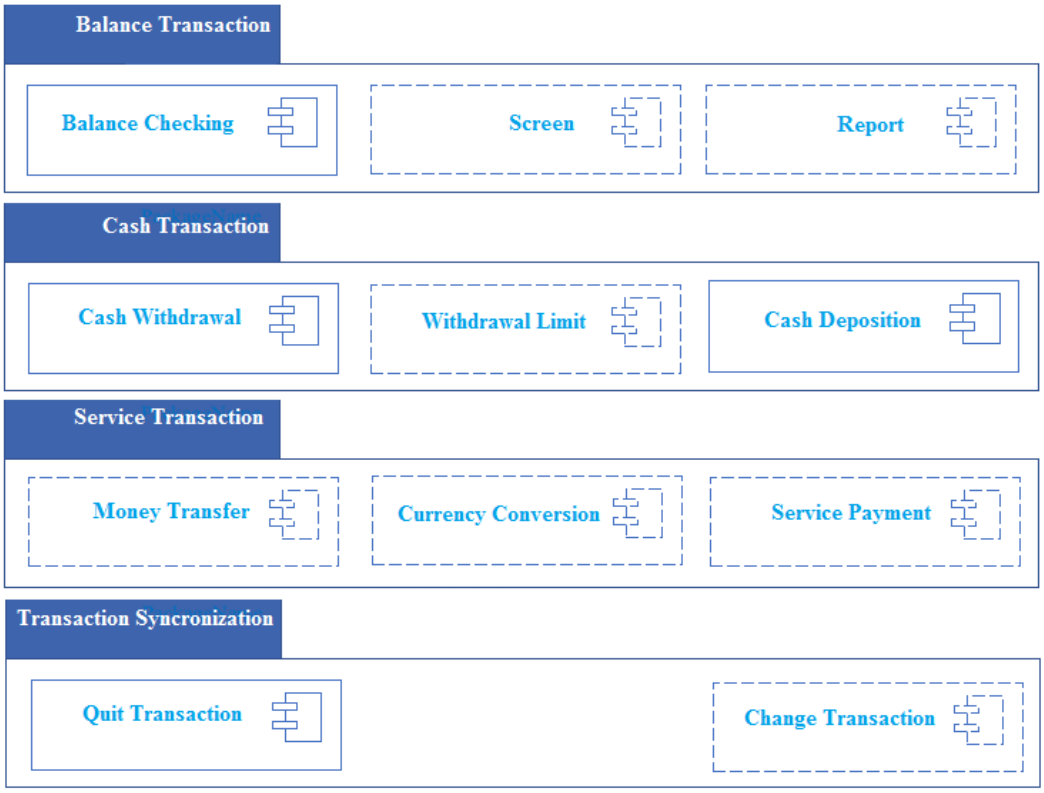


Figure 7.15: The 'ATM Transaction' most upper layer of the reference architecture.

The 'Balance Transaction' subsystem has the common 'Balance Checking' component for the common 'check_balance' feature and special components only for the variants. The 'check_balance' feature describes the way provided by the 'Balance Transaction' subsystem to check and show the balance. The variability in this subsystem offers an alternative with two variants that appear in the current FM as 'show_balance_screen' and 'show_balance_report'. Each of them is realised by a separate component inside the internal design of the 'Balance Transaction' subsystem. The former variant is realised by the 'Screen' separate component, and the latter is realised by the 'Report' separate component. Thus, the main objective of this example is to

show, in detailed steps, how to map features of the current FM to the corresponding feature-related-code fragments of the ATM resulting SPL, using the reference architecture shown in Fig. 7.14 and Fig. 7.15.

Feature- architecture mapping macro step

The feature-architecture mapping macro step follows the structure of the reference architecture (Fig. 7.14 and Figure Fig. 7.15) as a guidance to find the intended feature-related-arch-elements (feature-related-subsystem and feature-related-component) using the following steps.

Step 1.1 provides a traceability link between features of the current FM and the 'ATM Transaction' layer of the reference architecture. As shown in Fig. 7.16, the 'ATM Transaction' layer is the target and proper layer to perform the feature mapping.

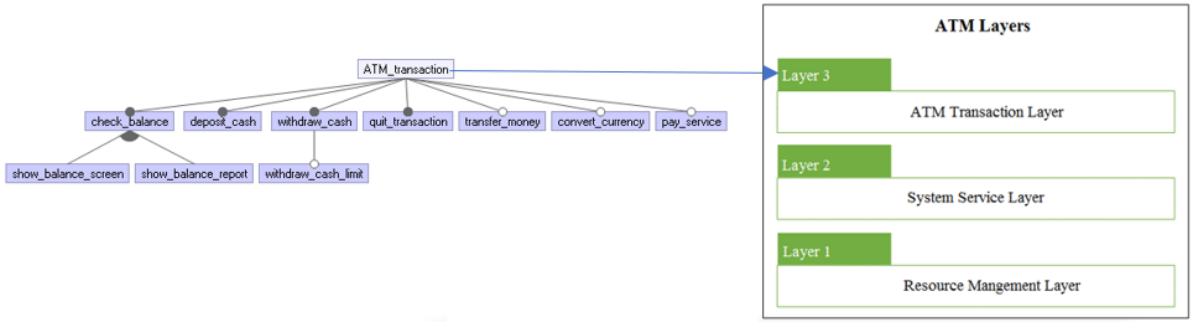


Figure 7.16: Traceability link between the features of the current FM and the 'ATM Transaction' most upper layer of the reference architecture.

Step 1.2 provides traceability links between features of the current FM and the feature-related subsystem of the 'ATM Transaction' layer.

Step 1.3 provides traceability links between each feature of the current FM and the feature-related component in the subsystem that is already identified for the feature, when required.

Step 1.2 and step 1.3 uses feature-requirements mapping, to assign feature_related_arch_elemts (feature-related subsystem and feature-related component) of each feature, as follows.

1. Identify the set of ARs of the feature from the FL.
2. Document the identified AR that belongs to the feature.

check_balance: {R2, R2.1, R2.2}

3. Relate each AR to its corresponding statement part from the RS document.

   R2: ATM checks the balance.
   R2.1: ATM shows the balance on screen.
   R2.2: ATM shows the balance on report.

4. Specify the functionality information of a set of ARs belongs to the feature.

5. Assign feature to feature-related-arch-elements.

   The feature represents a functionality related to balance checking.

Step 1.4 stores the traceability links of each feature to its feature-related-arch-elements in a document.

The 'check_balance' feature is related to:
'Balance Transaction' subsystem.
'Balance Checking' component.

Steps 1.1 to step 1.4 are explained using the 'check_balance' feature. The same process can be applied to all features of the current FM. Fig. 7.17 contains the traceability links from each feature of the current FM to feature-related-arch-elements.

The 'check_balance' feature is related to:
- 'Balance Transaction' subsystem.
- 'Balance Checking' component.

The 'show_balance_screen' feature is related to:
- 'Balance Transaction' subsystem.
- 'Screen' component.

The 'show_balance_report' feature is related to:
- 'Balance Transaction' subsystem.
- 'Report' component.

The 'withdrawal_cash' feature is related to:
- 'Cash Transaction' subsystem.
- 'Cash Withdrawal' component.

The 'withdrawal_cash_limit' feature is related to:
- 'Cash Transaction' subsystem.
- 'Withdrawal Limit' component

The 'deposit_cash' feature is related to:
- 'Cash Transaction' subsystem.
- 'Cash Deposit' component.

The 'transfer_money' feature is related to:
- 'Service Transaction' subsystem.
- 'Money Transfer' component.

The 'convert_currency' feature is related to:
- 'Service Transaction' subsystem.
- 'Currency Conversion' component.

The 'pay_service' feature is related to:
- 'Service Transaction' subsystem.
- 'Service Payment' component.

The 'quit_transaction' feature is related to:
- 'Transaction Synchronization' subsystem.
- 'Quit Transaction' component.

Figure 7.17: The document that contains the traceability links from each feature of the current FM to feature-related-arch-elements.

Table 7.1: Products of the ATM resulting SPL and the corresponding packages.

product name	main package	package corresponds to the 'ATM Transaction' layer
Product1	atmProduct1	atmTransaction
Product2	atmProduct2	atmTransaction
Product3	atmProduct3	atmTransaction
ProductNew	atmProductNew	atmTransaction

Table 7.2: Sub-packages of the ATM resulting SPL and the corresponding subsystems of the reference architecture.

sub-package name	subsystem name
balanceTransaction	Balance Transaction
cashTransaction	Cash Transaction
serviceTransaction	Service Transaction
transactionSynchronization	Transaction Synchronization

Feature-code mapping macro step

Step 1.5 investigates the code characteristics and the code variability nature of the products of the ATM resulting SPL.

The Evo SPL approach can deal with the code of the ATM resulting SPL, since it is satisfied the following characteristics.

- The code of each product is organized by a main package, which marks the product name, as well as it is composed of other packages, where each of them corresponds to a layer of the reference architecture. The 'atmTransaction' main package corresponds to the 'ATM Transaction' most upper layer of the reference architecture and has a unique and common name among all the products, as well as it has a name identifies close to the name of the 'ATM Transaction' layer of the reference architecture as shown in Table 7.1 .
- The 'atmTransaction' package has sub-packages. Each sub-package corresponds to a subsystem of the reference architecture, as shown in Table 7.2 .
- Each sub-package contains one or more modules, where each module has a specific functionality of the product and corresponds to a specific component of the reference architecture, as shown in Table 7.3.
- Each module contains one or more different source files that are shared to implement the functionality of the module and can be edited separately, as shown in Table 7.4.

Table 7.3: Sub-packages including the modules of the ATM resulting SPL and the corresponding components of the reference architecture.

sub-package name	module name	component name
balanceTransaction	balanceCheck	Balance Checking
	balanceScreen	Screen
	balanceReport	Report
cashTransaction	cashWithdrawal	Cash Withdrawal
	limitWithdrawal	Withdrawal Limit
	cashDeposit	Cash Deposition
serviceTransaction	transferMoney	Money Transfer
	convertCurrency	Currency Conversion
	payService	Service Payment
transactionSynchronization	quitTransaction	Quit Transaction
	changeTransaction	Change Transaction

Table 7.4: Modules of the of ATM resulting SPL and the source file of each module.

module name	source file name
balanceCheck	balanceCheck.cpp
balanceScreen	balanceScreen.cpp
balanceReport	balanceReport.cpp
cashWithdrawal	cashWithdrawal.cpp
limitWithdrawal	limitWithdrawal.cpp
cashDeposit	cashDeposit.cpp
transferMoney	transferMoney.cpp
convertCurrency	convertCurrence.cpp
payService	payService.cpp
quitTransaction	quitTransaction.cpp
changeTransaction	changeTransaction.cpp

- Each source file has a name that reflects its functionality and reflects the module and sub-package that the source file belongs to. The source files corresponding to internal design of the component. They are the target place of the feature-code mapping macro step, to search for the feature implementation.
- The implementation code elements follow the programming naming convention, which is a set of rules for choosing the character sequence to be used for identifiers that denote package, source file, routines, and other entities (local and global variables) in code and documentation. The ATM resulting SPL uses its reserved domain names for its package, source file, routine, and attribute names, which closely correlates with the reference architecture.

Step 1.6 creates five tracing rules that define how relationships can be established between features of the current FM and code fragments of the code. In other words, the rules restrict a feature of the current FM can be traced to which code fragment(s) of the code.

Tracing rule (1). All the features of the current FM can trace to the package containing the implementation of a specific product requirements inside the main package of a product or more (i.e., the package corresponds to the most upper layer of the reference Architecture) as follows.

The features of the current FM can trace to the 'atmTransaction' package.

Tracing rule (2). A feature can trace to the sub-package corresponding to the feature-related-subsystem, as presented in Tracing rule(2) title below.

Tracing rule(2)

1. The 'check_balance' feature can trace to the 'balanceTransaction' sub-package corresponding to the 'Balance Transaction' subsystem.
2. The 'show_balance_screen' feature can trace to the 'balanceTransaction' sub-package corresponding to the 'Balance Transaction' subsystem.
3. The 'show_balance_report' feature can trace to the 'balanceTransaction' sub-package corresponding to the 'Balance Transaction' subsystem.
4. The 'withdraw_cash' feature can trace to the 'cashTransaction' sub-package corresponding to the 'Cash Transaction' subsystem.
5. The 'withdraw_cash_limit' feature can trace to the 'cashTransaction' sub-package corresponding to the 'Cash Transaction' subsystem.

6. The ‘deposit_cash’ feature can trace to the ‘cashTransaction’ sub-package corresponding to the ‘Cash Transaction’ subsystem.
7. The ‘transfer_money’ feature can trace to the ‘serviceTransaction’ sub-package corresponding to ‘Service Transaction’ subsystem.
8. The ‘convert_currency’ feature can trace to the ‘serviceTransaction’ sub-package corresponding to ‘Service Transaction’ subsystem.
9. The ‘pay_service’ feature can trace to the ‘serviceTransaction’ sub-package corresponding to ‘Service Transaction’ subsystem.
10. The ‘quit_transaction’ feature can trace to the ‘transactionSynchronization’ sub-package corresponding to ‘Transaction Synchronization’ subsystem.

Tracing rule (3). A feature can trace to the specific module corresponding to the feature-related component in the subsystem that is already identified for the feature, as presented in Tracing rule(3) title below.

Tracing rule (3).

1. The ‘check_balance’ feature can trace to the ‘balanceCheck’ module corresponding to the ‘Balance Checking’ component in the ‘Balance Transaction’ subsystem that is already identified for the feature.
2. The ‘show_balance_screen’ feature can trace to the ‘balanceScreen’ module corresponding to the ‘Screen’ component in the ‘Balance Transaction’ subsystem that is already identified for the feature.
3. The ‘show_balance_report’ feature can trace to the ‘balanceReport’ module corresponding to the ‘Report’ component in the ‘Balance Transaction’ subsystem that is already identified for the feature.
4. The ‘withdraw_cash’ feature can trace to the ‘cashWithdrawal’ module corresponding to the ‘Cash Withdrawal’ component in the ‘Cash Transaction’ subsystem that is already identified for the feature.
5. The ‘withdraw_cash_limit’ feature can trace to the ‘limitWithdrawal’ module corresponding to the ‘Withdrawal Limit’ component in the ‘Cash Transaction’ subsystem that is already identified for the feature.

6. The ‘deposit_cash’ feature can trace to the ‘cashDeposit’ module corresponding to the ‘Cash Deposition’ component in the ‘Cash Transaction’ subsystem that is already identified for the feature.

7. The ‘transfer_money’ feature can trace to the ‘transferMoney’ module corresponding to the ‘Money Transfer’ component in the ‘Service Transaction’ subsystem that is already identified for the feature.

8. The ‘convert_currency’ feature can trace to the ‘convertCurrency’ module corresponding to the ‘Currency Conversion’ component in the ‘Service Transaction’ subsystem that is already identified for the feature.

9. The ‘pay_service’ feature can trace to the ‘payService’ module corresponding to the ‘Service Payment’ component in the ‘Service Transaction’ subsystem that is already identified for the feature.

10. The ‘quit_transaction’ feature can trace to the ‘quitTransaction’ module corresponding to the ‘Quit Transaction’ component in the ‘Transaction Synchronization’ subsystem that is already identified for the feature.

Tracing rule (4). A feature can trace to a set of source files related to the module that is already identified for the feature, as presented in Tracing rule (4) title below .

Tracing rule (4)

1. The ‘check_balance’ feature can trace to the ‘balanceCheck.cpp’ source file related to the ‘balanceCheck’ module that is already identified for the feature.

2. The ‘show_balance_screen’ feature can trace to the ‘balanceScreen.cpp’ source file related to the ‘balanceScreen’ module that is already identified for the feature.

3. The ‘show_balance_report’ feature can trace to the ‘balanceReport.cpp’ source file related to the ‘balanceReport’ module that is already identified for the feature.

4. The ‘withdraw_cash’ feature can trace to the ‘cashWithdrwal.cpp’ source file related to the ‘cashWithdrawal’ module that is already identified for the feature.

5. The ‘withdraw_cash_limit’ feature can trace the ‘limitWithdrawal.cpp’ source file related to the ‘limitWithdrawal’ module that is already identified for the feature.

6. The ‘deposit_cash’ feature can trace to the ‘cashDeposit.cpp’ source file related to the ‘cashDeposit’ module that is already identified for the feature.

7. The ‘transfer_money’ feature can trace to the ‘transferMoney.cpp’ source file related to the ‘transferMoney’ module that is already identified for the feature.

8. The ‘convert_currency’ feature can trace to the ‘convertCurrency.cpp’ source file related to the ‘convertCurrency’ module that is already identified for the feature.

9. The ‘pay_service’ feature can trace to the ‘payService.cpp’ source file related to the ‘payService’ module that is already identified for the feature.

10. The ‘quit_transaction’ feature can trace to the ‘quitTransaction.cpp’ source file related to the ‘quitTransaction’ module that is already identified for the feature.

Tracing rule (5). A feature can trace to a set of routines, statements, and expressions related to the source file that is already specified for the feature. This tracing rule can be applied when performing the feature location steps of step 1.7 below. Since, the source files have fine-grained implementation. They do not use routines, and they consist of statements, expressions, and attributes.

Step 1.7 investigates and locates the feature position in the code, using the following feature location steps.

1. Use Eclipse to import packages of the products of the resulting SPL into its local development workspace. Our approach does not require any amount of changes to the code.

2. Label each feature of the current FM with feature-related-product. As shown in Fig. 7.18, software engineers can follow one of the two strategies listed below when labelling each feature.

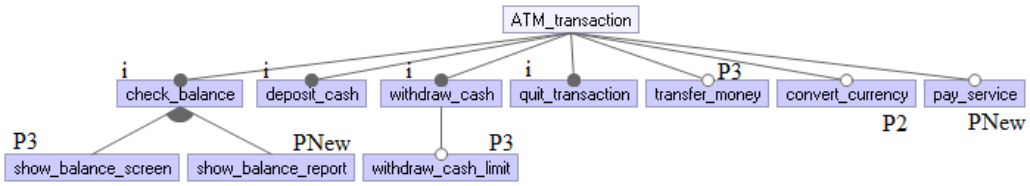


Figure 7.18: The current FM labelled with feature-related-products.

- Software engineers can label the common features with the initial release product (e.g., using ‘i’ character). For example, the common ‘check_balance’ feature is labelled with ‘i’ character. As presented earlier, in the illustrative example of the reverse engineering phase, Product 1 is the initial release product of the ATM products family.

- Software engineers can label an optional or alternative feature with one of the products that have (implements) this feature. For example, the optional 'convert_currency' feature is labelled with its feature-related product 'P2', where the alternative 'show_balance_screen' feature is labelled with its feature-related-product 'P3'.

3. Use the current FM, which has been labelled with feature-related-product (see Fig. 7.18) and the traceability links that are established as a result of the tracing rules of step 1.6, to find the feature positions (feature-related-code fragments) in the code of the feature-related-product.

4. Prepare the feature-related-terms. Software engineer prepare and review a set of terms related to each feature from artifacts of the feature-related product, like the manual guide, requirements document as well as RS (e.g., ARs may contain meaningful terms), and code (e.g., code comments may contain meaningful terms) of the code. For example, a set of predefined feature-related-terms may include the feature name itself, technology, signal, service, capability, and attribute (see Table 7.5 and Table 7.6).

5. Search the code of the feature-related- product for the feature-related-terms and retrieve the corresponding feature-related-code-fragments. The search process targets the code in the source files of the feature-related-product labelled in the current FM. The search process uses the search capability of the Eclipse, to locate and return feature-related-code fragments. The Eclipse IDE search capability highlights the feature-related-code fragments using code visualization capability of the same IDE (see Eclipse). As there is no enough place, on the following, we list some features and the related feature-code fragments (see Table 7.7 Table 7.8).

6. Relate the feature with feature-related-code fragments, once the matching appears, by highlighting the places of the code that matched the feature- related-terms (see Fig. 7.19).

Step 1.8 creates the traceability links from the feature to its positions in the code. Each traceability link represents a relation between a feature and the set of code fragments implementing the feature.

Step 1.9 stores the traceability links in the traceability tree, as shown in Fig. 7.20.

Repeats step 1.7 to step 1.9 until all the features of the current FM are considered.

Feature mapping conformance macro step

Table 7.5: Feature-related-terms of the ATM resulting SPL.

feature	feature-related -product	source	feature-related -terms
check_balance	Product 1	manual guide	not exist
		requirements document	R2=ATM checks the balance. {ATM, check, balance}
		code	e.g., //show check balance option on screen {check, balance}
show_balance_screen	Product 3	manual guide	not exist
		requirements document	R2.1=ATM shows the balance on screen {ATM, show, balance, screen}
		code	e.g., //show the balance on screen {show, balance, screen}
show_balance_report	ProductNew	manual guide	not exist
		requirements document	R2.2=ATM shows the balance on report. {ATM, show, balance, report}
		code	e.g., //show the balance on report {show, balance, report}
withdraw_cash	Product 1	manual guide	not exist
		requirements document	R3 = ATM withdraws cash. {ATM, withdraw, cash}
		code	e.g., //show withdraw cash option on screen e.g., //enter the amount to withdraw e.g., // define long withdraw; {withdraw, cash}
withdraw_cash_limit	Product 3	manual guide	not exist
		requirements document	R3.1=ATM withdraws cash with a limit. {ATM, withdraw, cash, limit}
		code	e.g., //show a message to withdraw cash with a limit {ATM, withdraw, cash, limit}
deposit_cash	Product1	manual guide	not exist
		requirements document	R4 = ATM deposits cash. {ATM, deposit, cash}
		code	e.g., //show the deposit cash option on screen e.g., //enter the amount to deposit e.g., // define long deposit; {deposit, cash}

Table 7.6: Feature-related-terms of the ATM resulting SPL continued.

feature	feature-related -product	source	feature-related -terms
transfer_money	Product 3	manual guide	not exist
		requirements document	R5 = ATM transfers money. {ATM, transfer, money}
		code	e.g., //show the transfer money option on screen e.g., //enter the amount to transfer e.g., // define long amount; {transfer, money, amount}
quit_transaction	Product 1	manual guide	not exist
		requirements document	R6 = ATM quits the transaction. {ATM, quit, transaction}
		code	e.g., //show the quit transaction message {quit, transaction}
convert_currency	Product 2	manual guide	not exist
		requirements document	R7 = ATM converts between currencies. {ATM, convert, currencies}
		code	e.g., //show convert currencies option on screen e.g., // define long currency; {convert, currency}
pay_service	ProductNew	manual guide	not exist
		requirements document	R8 = ATM pays for services. {ATM, pay, service}
		code	e.g., //show pay services option on screen e.g., // define char serviceType [10]; {pay, service}

Table 7.7: Feature-related-code fragments of the ATM resulting SPL.

feature: feature-related -terms	feature-related -product
check_balance: {ATM, check, balance}	Product 1
feature-related-code fragments : code lines in source file	

1. cod-fragment 1: code lines 19-22
do { print f (”********Welcome to ATM
Service**********\n”);
printf(”1. Check Balance\n”);
2. cod-fragment 2: code lines 27-33
printf(”Enter your choice: ”);
scanf(”%d”, &choice);
switch (choice){
case 1:
printf(”\n YOUR BALANCE IN Rs : %lu ”, amount);
break;
3. code-fragment 3: code lines 41-55
else if (withdraw >(amount - 500)) {
printf(”\n INSUFFICENT BALANCE”);}
else {amount = amount - withdraw;
printf(”\n\n PLEASE COLLECT CASH”);
printf(”\n YOUR CURRENT BALANCE IS%lu”, amount);}
4. code-fragment 4: code line 56
printf(”YOUR BALANCE IS %lu”, amount);	

feature: feature-related -terms	feature-related -product
Withdraw_cash:{ATM, withdraw, cash}	Product 1
feature-related-code fragments : code lines in source file	

1. cod-fragment 5: code line 6
unsigned long amount=1000, deposit, withdraw;
2. code-fragment 6: code lines 19-23
do{printf (”********Welcome to ATM Service**********\n”);
printf(”2. Withdraw Cash\n”);
3. code-fragment 7: code lines 34-51
case 2:
printf(”\n ENTER THE AMOUNT TO WITHDRAW: ”);
scanf(”%lu”, &withdraw);
if (withdraw % 100 != 0){
printf(”\n PLEASE ENTER THE AMOUNT IN MULTIPLES OF 100”);
}else if (withdraw >(amount - 500)){
printf(”PLEASE COLLECT CASH”);
printf(”YOUR CURRENT BALANCE IS %lu”, amount);}
break;	

Table 7.8: Feature-related-code fragments of the ATM resulting SPL continued.

feature: feature-related-terms	feature-related-product
Deposit_cash:{ATM, deposit, cash}	Product 1
Feature-related-code fragments : code lines in source file	

1. code-fragment 8: code line 6
unsigned long amount=1000, deposit, withdraw;
2. code-fragment 9: code lines 19-24
do{printf("********Welcome to ATM Service**********\n");
printf("3. Deposit Cash\n");
3. code-fragment 10: code lines 52-57
case 3:
printf("\n ENTER THE AMOUNT TO DEPOSIT");
scanf("%lu", &deposit);
amount = amount + deposit;
printf("YOUR BALANCE IS %lu", amount);
break;	

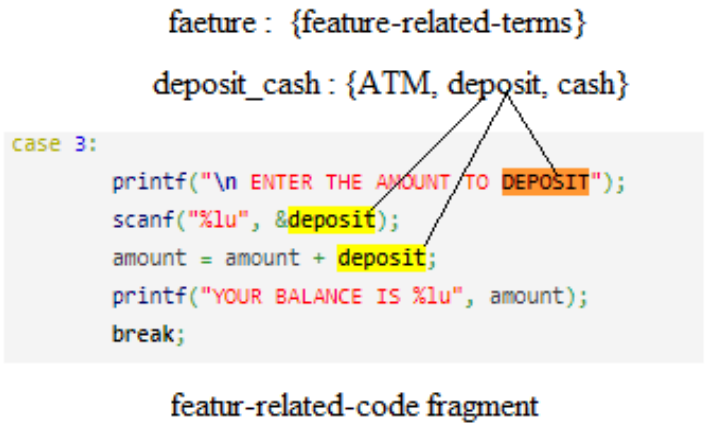


Figure 7.19: Highlighting the places of the code that matched the feature-related-terms.

ATM resulting SPL : Traceability tree	
feature	**feature-related-code fragments**
check_balance	code-fragment 1, code-fragment 2, code-fragment 3, and cod-fragment 4.
withdrawal_cash	code-fragment 5, code-fragment 6, and code-fragment 7.
deposit_cash	code-fragment 8, and code-fragment 9, code-fragment 10.

Figure 7.20: Traceability tree of the ATM resulting SPL.

The feature mapping conformance macro step maintains the current FM and the code conformance of the ATM resulting SPL, whenever feature changes occur (e.g., add new feature). This

macro step can (automatically) update the traceability links of the traceability tree (Fig. 7.20), to maintain the coherence between features and code fragments, when existing features are changed.

The feature mapping conformance macro step works on the already established traceability links of the traceability tree. It works in case of features change occur in the current FM. Thus, it applies predefined steps for each case as follows.

Case 1 (add feature). When a feature change occurs to the current FM, which involves adding the 'change_transaction' and 'show_message' features, then the feature mapping conformance macro step (1) adds the 'change_transaction and 'show_message' features to the features level of the traceability tree, (2) gives the new features a red color, and (3) creates new links from the new features to feature-related-code fragments, as shown in Fig. 7.21.

ATM resulting SPL : Traceability tree	
feature	**feature-related-code fragments**
check_balance	code-fragment 1, code-fragment 2, code-fragment 3, and cod-fragment 4.
withdrawal_cash	code-fragment 5, code-fragment 6, and code-fragment 7.
deposit_cash	code-fragment 8, and code-fragment 9, code-fragment 10.
change_transaction	code-fragment 11, and code-fragment 12.
show_message	code-fragment 13.

Figure 7.21: Traceability tree of the ATM resulting SPL after adding 'change_transaction' and 'show_message' features.

Case 2 (delete feature). When a feature change occurs to the current FM , which involves deleting the 'show_message' feature, then the feature mapping conformance macro step deletes this feature from the features level of the traceability tree, and (2) removes all the tractability links from this feature to feature-related code fragments of the traceability tree, as shown in FigureFig. 7.22 and Fig. 7.23.

7.3 Tool support: friendlyMapper

This section presents the friendlyMapper tool, which supports the feature mapping activity presented in the mapping phase of the EvoSPL approach [Fri]. In specific, it supports the feature-code mapping and feature mapping conformance macro steps. The tool supports a set of tasks that can be used by software engineers to create, store, and maintain the relationships between

ATM resulting SPL : Traceability tree	
feature	**feature-related-code fragments**
check_balance	code-fragment 1, code-fragment 2, code-fragment 3, and cod-fragment 4.
withdrawal_cash	code-fragment 5, code-fragment 6, and code-fragment 7.
deposit_cash	code-fragment 8, and code-fragment 9, code-fragment 10.
change_transaction	code-fragment 11, and code-fragment 12.
~~show_message~~	~~code-fragment 13.~~

Figure 7.22: Traceability tree of the ATM resulting SPL when preparing to delete 'show_message' feature.

ATM resulting SPL : Traceability tree	
feature	**feature-related-code fragments**
check_balance	code-fragment 1, code-fragment 2, code-fragment 3, and cod-fragment 4.
withdrawal_cash	code-fragment 5, code-fragment 6, and code-fragment 7.
deposit_cash	code-fragment 8, and code-fragment 9, code-fragment 10.
change_transaction	code-fragment 11, and code-fragment 12.

Figure 7.23: Traceability tree of the ATM resulting SPL after deleting 'show_message' feature.

features belonging to the current FM and feature-related-code fragments (i.e., routines) of the code, using the traceability tree. The tool includes modelling and visualizing the traceability tree, which consists of traceability links that relate a feature to the feature-related-code fragments (i.e., routines).

Fig. 7.24 is a screenshot of the friendlyMapper tool. It shows an example of ATM resulting SPL presented in section 7.2. The figure shows three windows: (1) Traceability tree, (2) Routine information, and (3) Routine list. The first window (Traceability tree) shows the features of the current FM taken directly from the XML file related to a feature tree, by selecting 'Import features' when showing a context menu, upon right clicking on the 'Feature ' item (see Fig. 7.25).

The second window (Routine information) presents a window that allows software engineers to enter the routines information of the resulting SPL (e.g., code-fragments of the ATM resulting SPL), by selecting 'Routine information' item when showing the context menu, upon right clicking on the resulting SPL package (e.g., atmTransaction), as shown in Fig. 7.26. The 'Routine

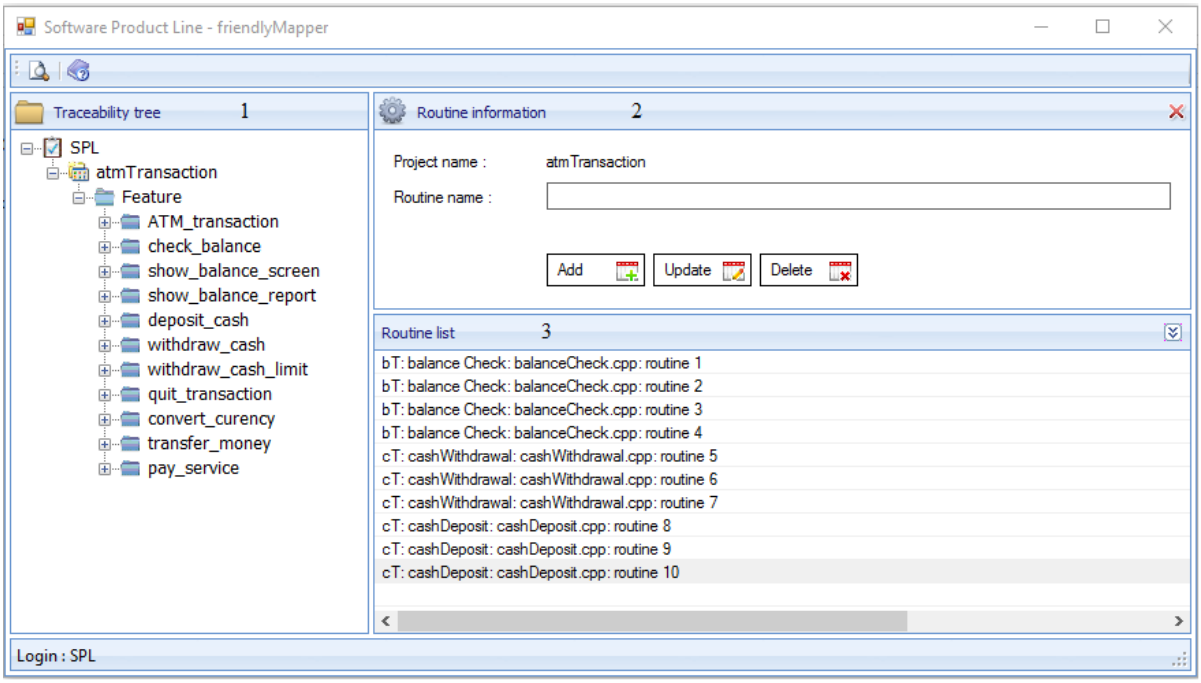


Figure 7.24: Screenshot for the main windows of friendlyMapper.

information' window allows software engineers to add, update and delete a routine (see window 2 of Fig. 7.24).

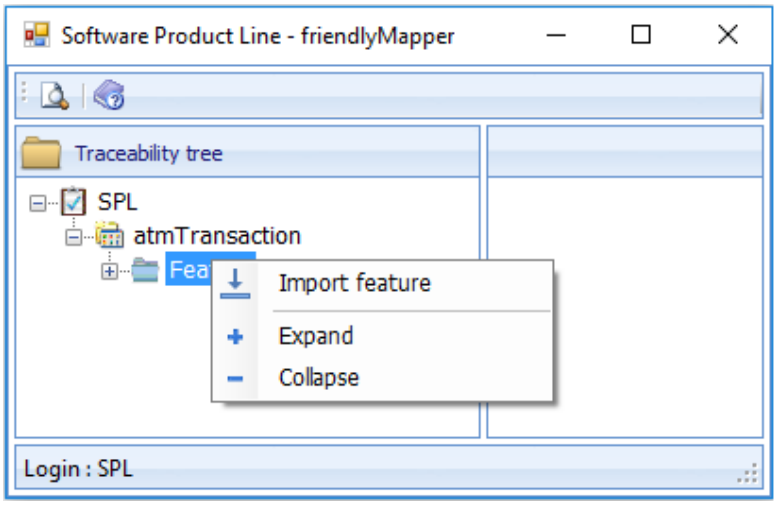


Figure 7.25: Import features of the current FM from XML file in friendlyMapper tool.

To add a routine, software engineers can enter the routine information (i.e., sub-package: module: source file.cpp: routine name) in the text field labelled with 'Routine name' text, and then they can press the 'Add' button associated with window 2 (Routine information) of the tool. To update a routine name software engineers can select routine from the list labelled with 'Routine

list' text and edit the routine name, when it appeared in the text field labelled with 'routine name' text, and press the 'Update' button associated with window 2 (Routine information) of the tool. To delete a routine from the list, software engineers can select routine from the list labelled with 'Routine list' text and then press the 'delete' button associated with window 2 (Routine information) of the tool. In consequence, the routine will be removed from the routine list.

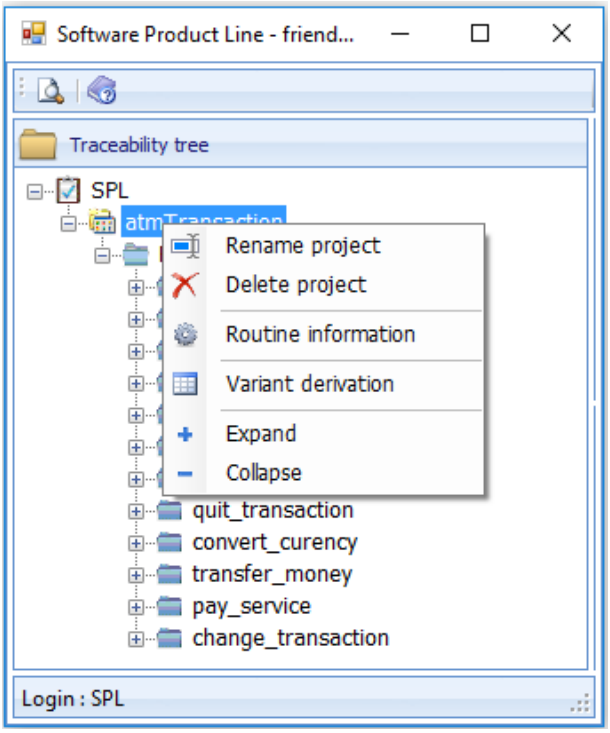


Figure 7.26: Add routine information of the code in the friendlyMapper tool.

The third window (Routine list) in the figure shows the list of routines of the resulting SPL that are responsible for a feature implementation in the code. There is another window related to this window (Routine list), called the 'Feature routine information'. Fig. 7.24 depicts the fourth window (Feature routine information), which appears by selecting the 'Add routine' item from the context menu (see Fig. 7.28), when right clicking on 'Routine' sub-branch of a feature (e.g., check_balance). This window allows software engineers to select a routine that is related to a specific feature (e.g., check_balance) from the left-hand side list and to add it to the right-hand side list, using the buttons occupied in the middle position of the two lists . At the same time, the routine will be added to 'Routine' sub-branch of the specified feature on the feature level (e.g., check_balance), as shown in Fig. 7.27.

Traceability tree of the friendlyMapper The tool uses traceability tree to create and update the traceability links whenever feature changes occur. The traceability is defined as one-to-many

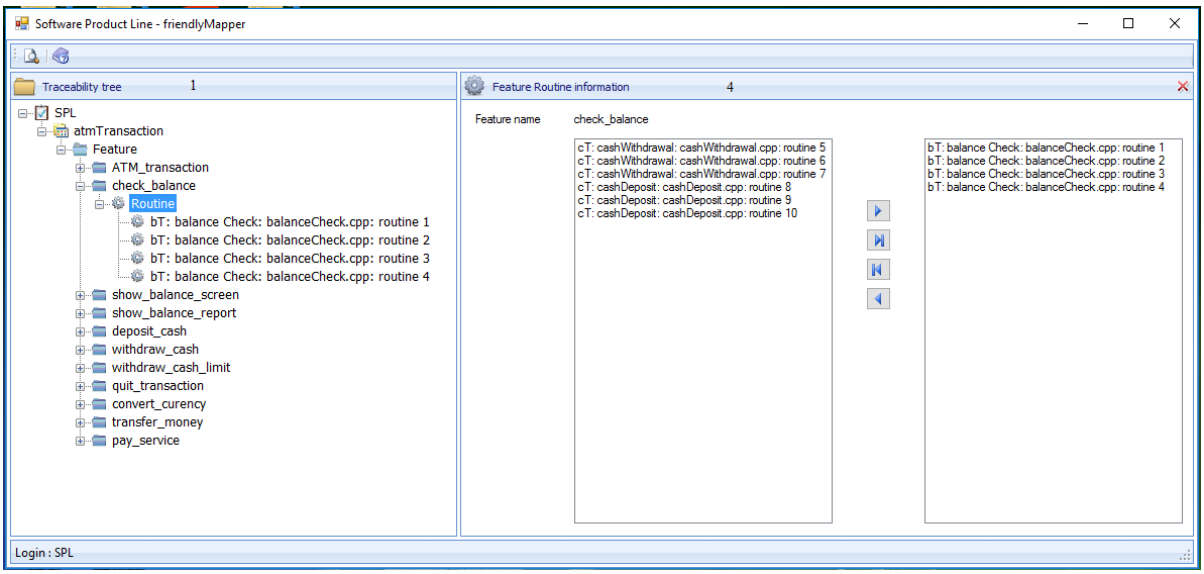


Figure 7.27: Feature routine information window of the friendlyMapper tool.

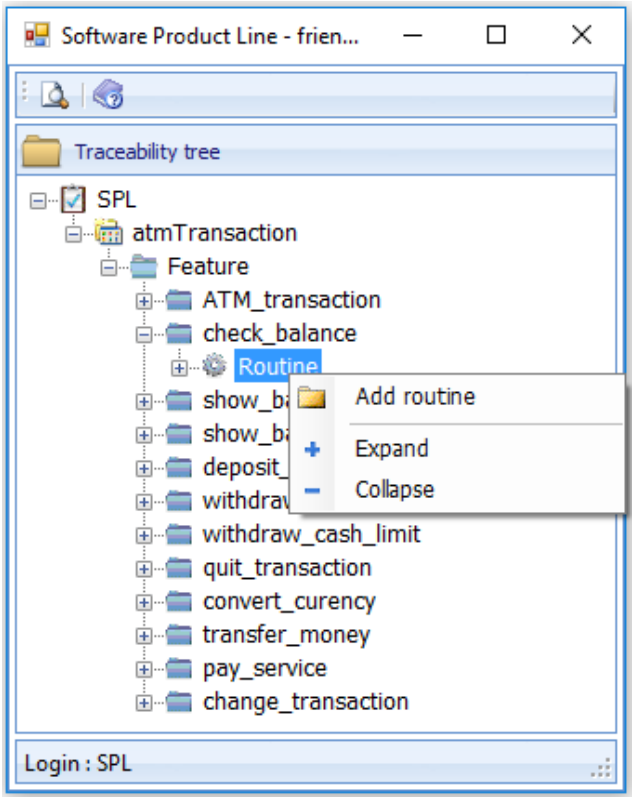


Figure 7.28: Add routine to a specific feature in friendlyMapper tool.

relation (a feature is related to a routine or more). Traceability tree requires (1) features of the current FM (taken from XML file) and (2) routine information of the code. The tool allows software engineers to use traceability tree to store and maintain the relationships from a feature

to the routines. The tool semi-automatically manages the relationship between each feature of the current FM and routines corresponding to the feature in the code.

As shown in window 1 of Fig. 7.27, traceability tree is a kind of dropdown tree starting its third level at 'Feature' item. When dropping down the 'Feature' level, the list of features (which are imported from the XML file that represents the current FM) appears sequentially. Each feature has a sub-branch, called 'Routine', which show routines that are related to the feature. Traceability tree maps artifacts of the resulting SPL (the current FM and code) to each other. It allows software engineers to trace a feature of the current FM to routines just by dropping down each level, starting at SPL project, the resulting SPL package, features, and moving toward routines of the selected feature (e.g., check_balance).

Routine information As explained earlier in this chapter a routine is one of the code elements (software units) that may implement a feature in the feature mapping activity. In addition, the tracing rules define how to relate features belonging to the current FM to feature-related-code fragments (e.g., routines) belonging to the code. FriendlyMapper introduces feature-related-code fragment as a code portion that maintains feature implementation, which is defined in the tool as a routine.

In friendlyMapper, the traceability tree allows software engineers to trace a feature to routines with their positions in the code, just by defining the routine location in the resulting SPL main package, starting at sub-package, module, source file, and moving toward routine name (i.e., sub-package: module: soucefile.cpp: routine), but it ignores attributes, statements and expressions (see 'Routine list' of window 3 in Fig. 7.24 and Fig. 7.27).

7.3.1 Tool capability

As mentioned earlier in this section, friendlyMapper tool is implemented in this thesis to support the feature mapping activity and its related macro steps (i.e., feature-code mapping and feature mapping conformance macro steps). Thus, this subsection presents the main capabilities of the tool according to them, as follows.

- Feature-code mapping macro step: to support this macro step, the tool is able to perform the following.
    1. Create a new SPL project, by selecting the 'new project' item from the context menu, as depicted in the right-hand side window of Figure 7.28, when right clicking on 'SPL' item (first level of the traceability tree), as depicted in the left-hand side window of Fig. 7.29.

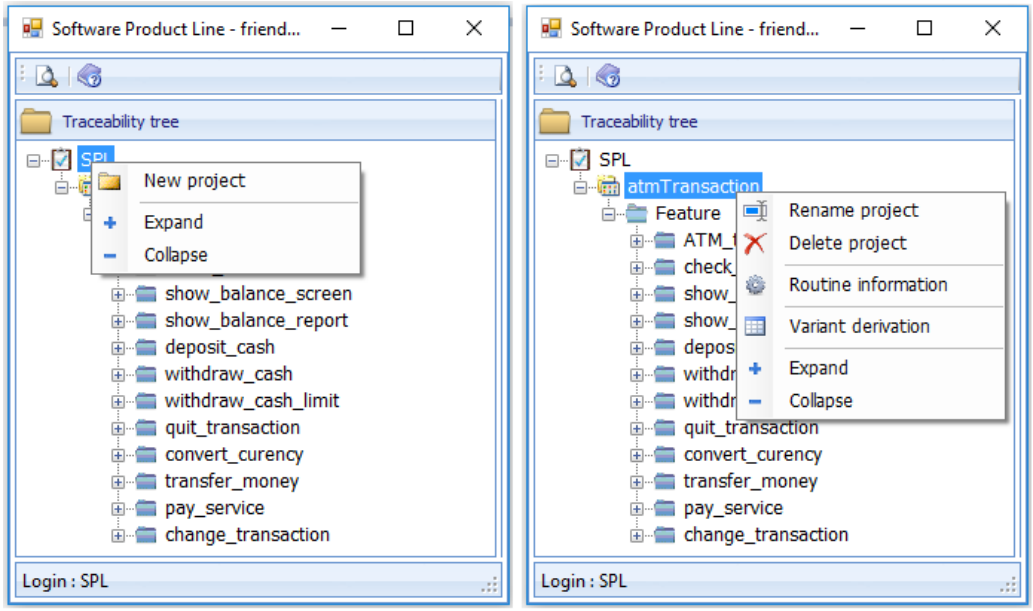


Figure 7.29: Sample example of creating and deleting an SPL project in friendlyMapper.

2. Remove an SPL project, by selecting 'delete project' item from the context menu, when right clicking on the project name (e.g., atmTransaction - second level of the traceability tree), as depicted in the right-hand side window of Fig. 7.29.
3. Import features of the current FM modelled a given resulting SPL, from the XML file written in a specific format, as shown in Fig. 7.30.

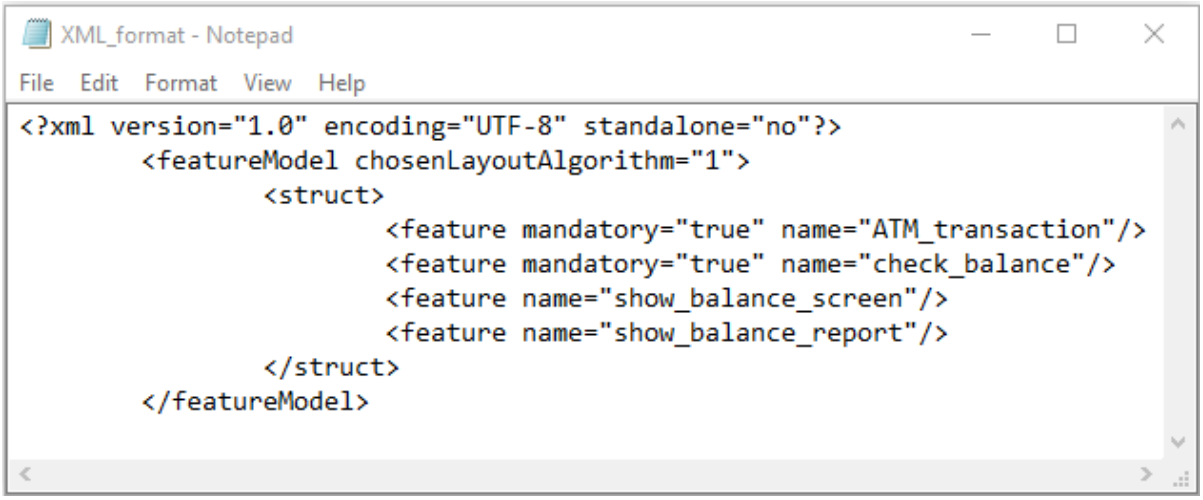


Figure 7.30: The XML file format related to the current FM and accepted by friendlyMapper.

4. Create a features list of the traceability tree under the 'Feature' item (on the 'Feature' level), as depicted in Fig. 7.25. The result of the features list of the traceability tree is depicted in window 1 of Fig. 7.24.

5. Create a traceability link from a feature (e.g., check_balance) of the features list to one or more routine (e.g., bT: balance Check: balanceCheck.cpp: routine 1), by selecting 'Add routine' item of the context menu, when selecting the feature (see Fig. 7.27).

- feature mapping conformance macro step: to support this macro step, the tool is able to perform the following.

1. Update automatically the traceability tree and routine information of the related screens (windows), whenever changes occur on all the levels (project, features, and subroutines) of its drop-down tree, which ensures that the features and code are consistent.

   For example, software engineers can relate an existing routine to a selected feature and the traceability tree automatically shows the new traceability link in its tree. Also, software engineers can add new routines to 'Routine lists', and the tool automatically updates 'Routine lists' in window 2 (Routine information) and window 4 (Routine feature information) of the tool (see Fig. 7.24 and Fig. 7.27) with new routines.

2. Update the features list of the traceability tree with new features whenever a change occurs to the current FM, by enabling software engineer to (i) re-import the features again, by reading the XML file contained the updated features and (ii) give the new features a red color, to alert software engineers about the existence of the new features (e.g., change_transaction and show_message), as shown in Fig. 7.31.

3. Remove a feature from the feature list of the traceability tree, by selecting 'Delete feature' from the context menu, when right clicking on the feature name. The feature will be removed from the features list upon approving the deletion from the confirmation dialog box, as shown in Fig. 7.32.

4. Add a new traceability link of the traceability tree, by first selecting the feature, opening its sub-branch - it is indicated with a plus sign (+) in front of the feature name, and selecting 'Add routine' item from the context menu, upon right clicking 'Routine' sub-branch of the selected feature. Once window 4 (Feature information routine) appeared, select the routine from the left- hand-side text and move it to right-hand side text, using the buttons occupied in the middle position of the sides (see Fig. 7.28, Fig. 7.27, and Fig. 7.33).

5. Import new features with a red color, to avoid software engineers from missing updating the code (see Fig. 7.31) .

   As shown in Fig. 7.33, the new features of the traceability tree are imported in a red color. This enables developers to confirm that they update routines of the code and

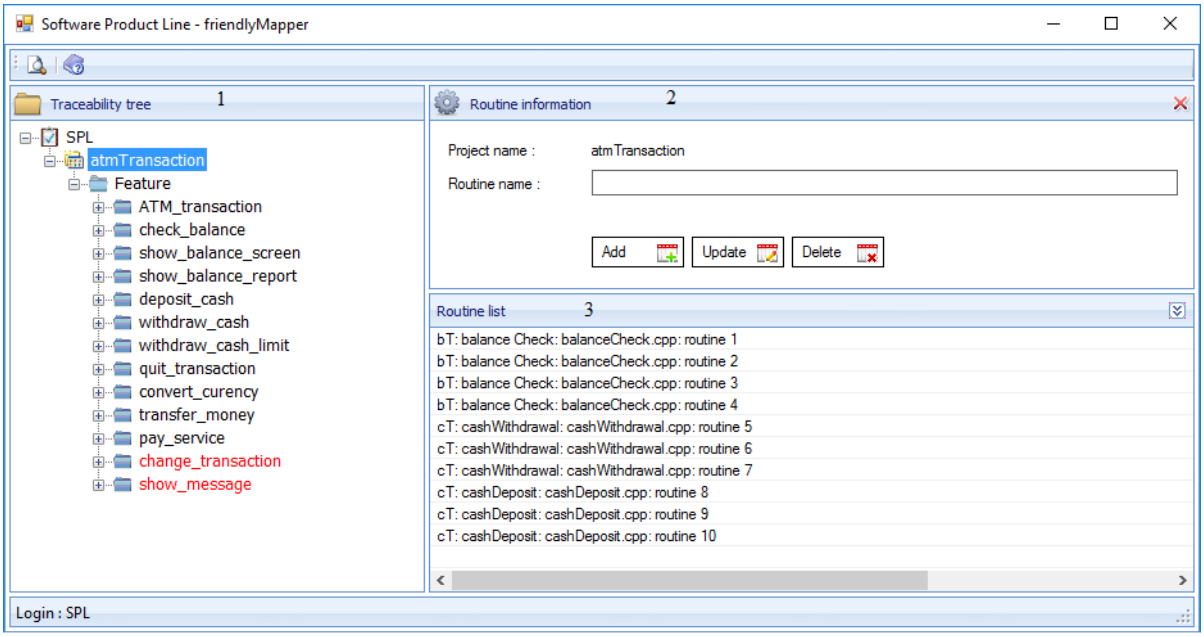


Figure 7.31: Features list of the traceability tree after re-importing features from the XML file including new ones.

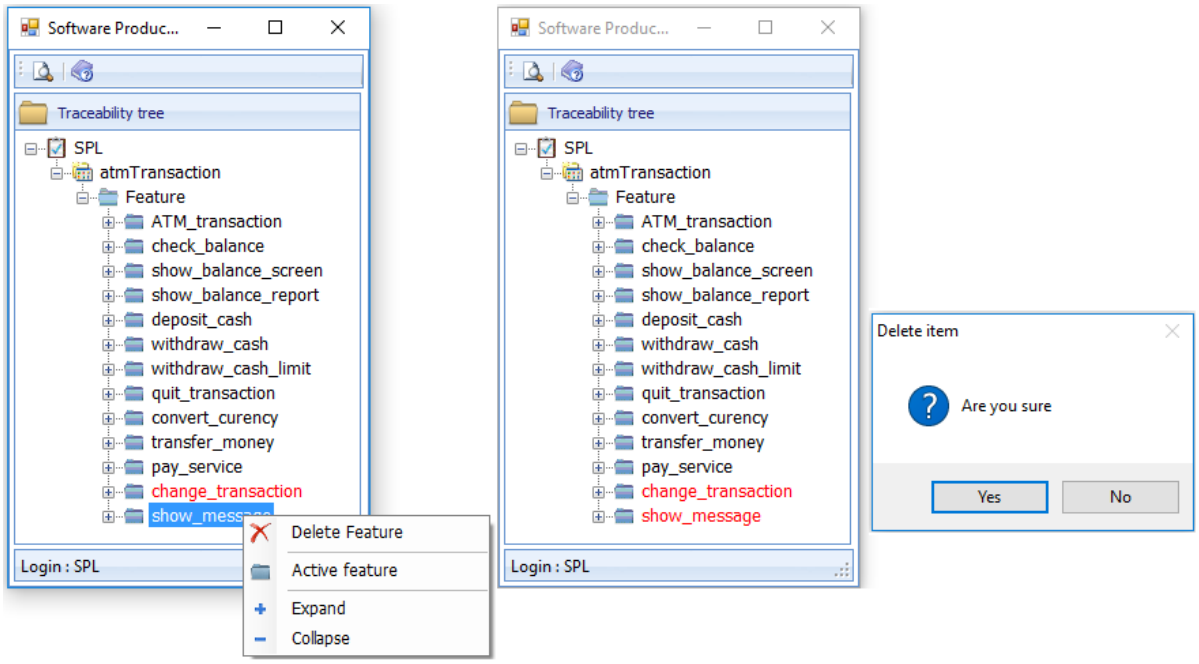


Figure 7.32: Delete 'show_message' feature from the traceability tree.

provide a feedback on changes made, which helps to preserve consistency between the current FM and code of the resulting SPL.

6. Change the color of a new feature (e.g., change_transaction) from red to color of the remaining features in the list of traceability tree, by selecting 'Activate feature'

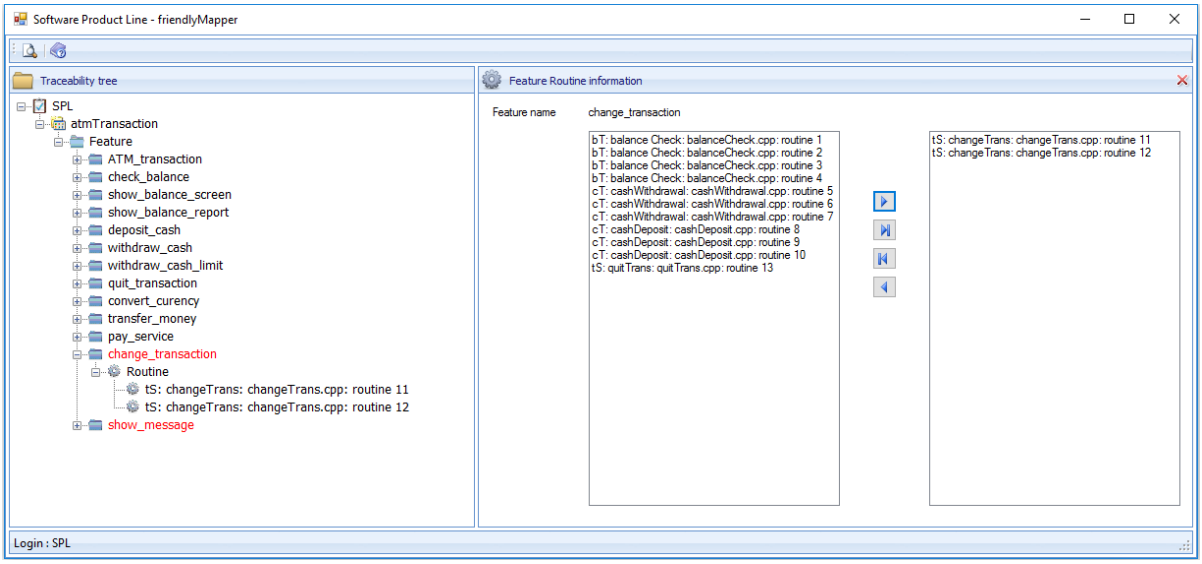


Figure 7.33: Traceability tree after adding a new traceability link from the 'change_transaction' feature to its corresponding routines.

from the context menu, when right clicking the feature. Fig. 7.34 shows an example of changing the color of 'change_transaction' feature from red color (left-hand side window) to color (i.e., black) of the features in the list of 'atmTransaction' traceability tree (right-hand side window), by select 'Active feature', when right clicking on the feature (middle-side window).

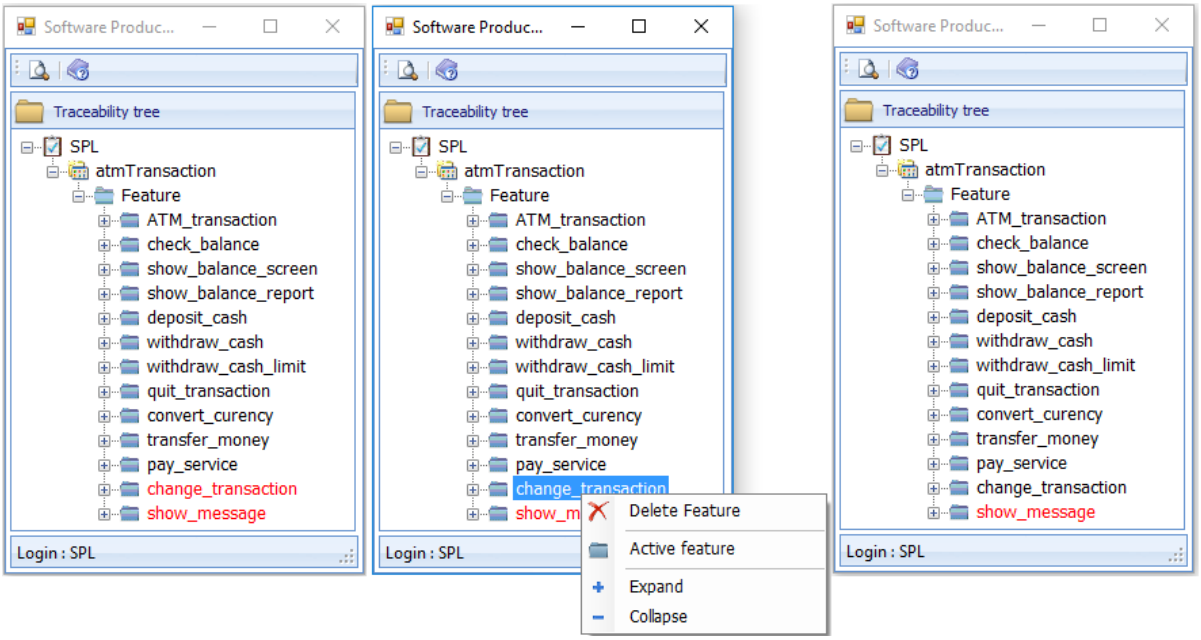


Figure 7.34: Change color of the 'change_transaction' feature from red color to black color.

7.3.2 An illustrative example: ATM products family: tool support

This subsection introduces the friendlyMapper tool for the same illustrative example presented in this chapter and the earlier chapters of our thesis, including this chapter. Given the ATM products family depicted in the lower side of Fig. 7.13 and the current FM modelled ATM resulting SPL depicted in the upper side of Fig. 7.13 and the XML file repents the model, as shown in Fig. 7.35. Additionally, taken into consideration Table 7.1 to Table 7.6 as well as Tracing rule (1) to Tracing rule (5) titles, the tool supports feature-code mapping macro step (starting at step 1.8) and feature mapping conformance macro step as follows.

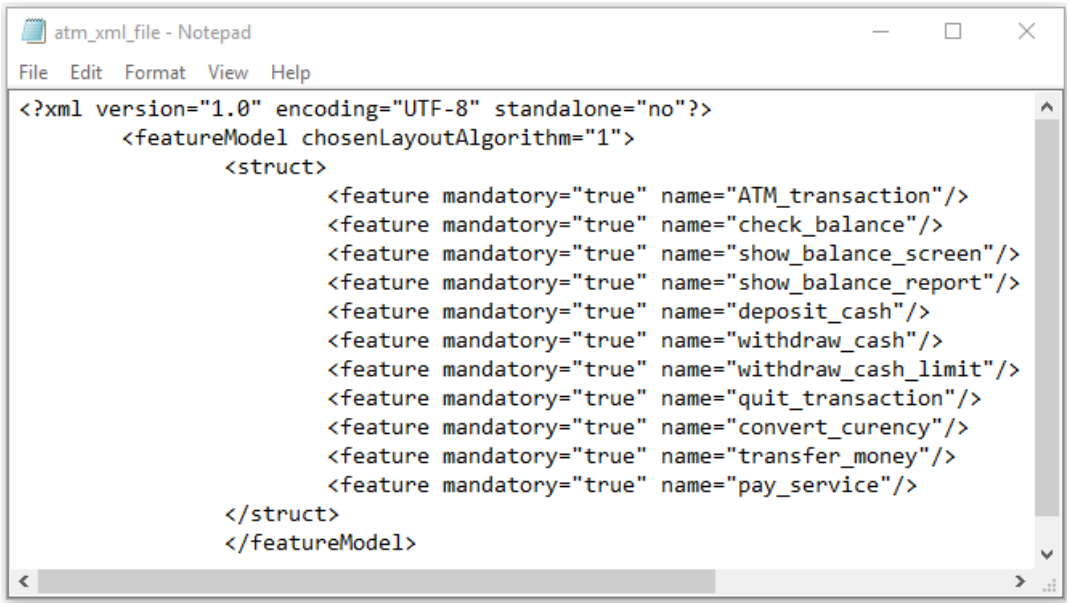


Figure 7.35: The XML file represents the ATM current model.

- Feature-code mapping macro step: to support this macro step, software engineer can use the tool to perform the following.
  1. Create a new SPL project, give the project 'atmTransaction' title, and import features of the current FM from the XML file, as shown in Fig. 7.35 and Fig. 7.36.
  2. Add routines of the ATM resulting SPL (feature-related-code fragments), as listed in Table 7.7 and Table 7.8. Software engineers can right click the 'atmTransaction' item, choose 'Routine information' menu item from the context menu (see Fig. 7.37), enter the routine information (i.e., sub-package: module: source file.cpp: routine name) in the text field labelled with 'Routine name' text, and then press the 'Add' button associated with window 2 (Routine information) of the tool (see Fig. 7.24). For routines positions of the 'atmTransaction' package, see Table 7.2 to Table 7.4.
  3. Create traceability links of the traceability tree as follows.

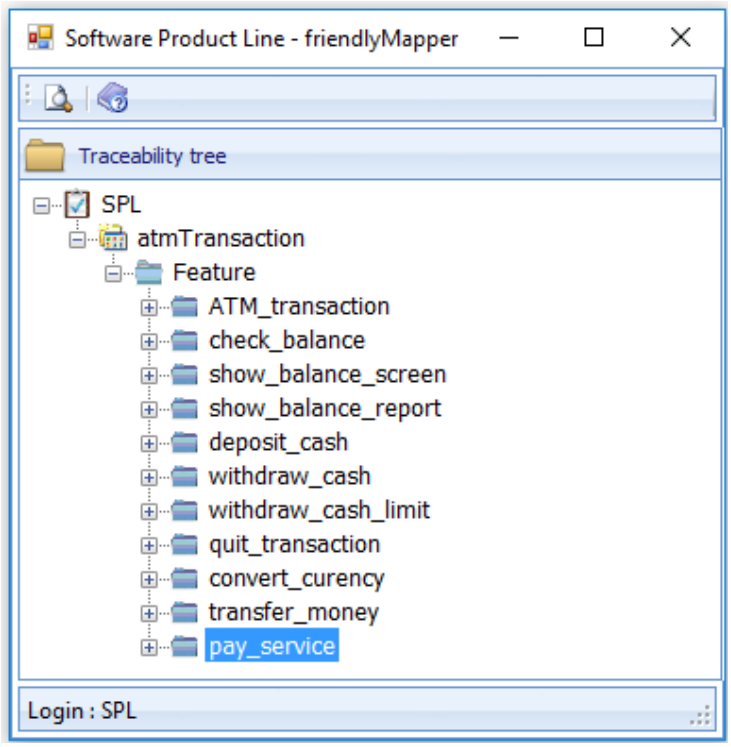


Figure 7.36: Features list of the ATM resulting SPL.

I Create a traceability link from the 'check_balance' feature to (1) bT: balance Check: balanceCheck.cpp: routine 1, (2) bT: balance Check: balanceCheck.cpp: routine 2, and (3) bT: balance Check: balanceCheck.cpp: routine 3, by selecting 'Add routine' item of the context menu, upon right-clicking on the 'Routine' sub-branch of the 'check_balance' feature (see Fig. 7.27). The traceability links from the 'check_balance' feature to its related routines, using the tool, correspond to the traceability links appearing in row 3 of Fig. 7.20.

II Create a traceability link from the 'withdraw_cash' feature to (1) cT: cashWithdrawal: cashWithdrawal.cpp: routine 5, (2) cT: cashWithdrawal: cashWithdrawal.cpp: routine 6, and (3) cT: cashWithdrawal: cashWithdrawal.cpp: routine 7, by selecting 'Add routine' item of the context menu, upon right-clicking on the 'Routine' sub-branch of the 'withdraw_cash' feature (see Fig. 7.38). The traceability links from the 'withdraw_cash' feature to its related routines, using the tool, correspond to the traceability links appearing in row 4 of Fig. 7.20.

III Create a traceability link from the 'deposit_cash' feature to (1) cT: cashDeposit: cashDeposit.cpp: routine 8, (2) cT: cashDeposit: cashDeposit.cpp: routine 9, and (3) cT: cashDeposit: cashDeposit.cpp: routine 10, by selecting 'Add routine' item of the context menu, upon right-clicking on the 'Routine' sub-branch of the 'deposit_cash' feature (see Fig. 7.39). The traceability links from the

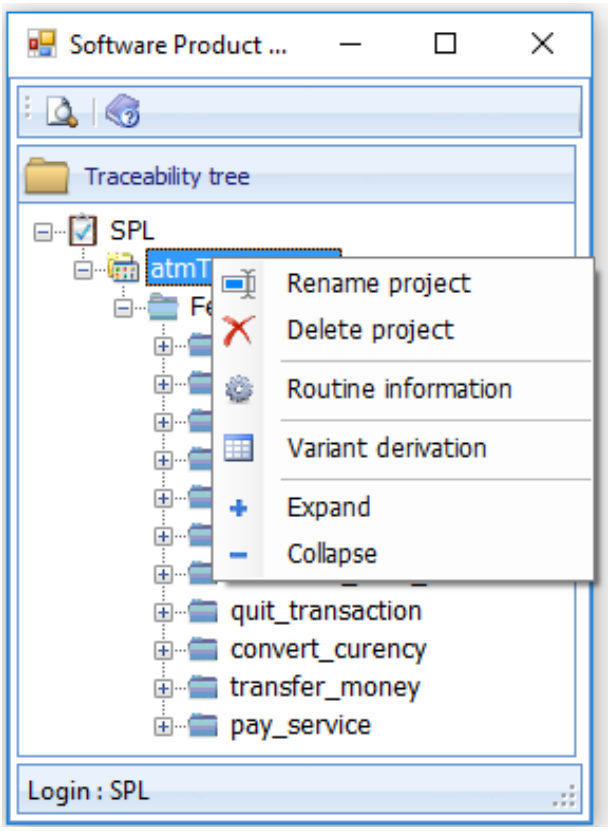


Figure 7.37: Add routines using the ‘ Routine information’ menu item of the context menu.

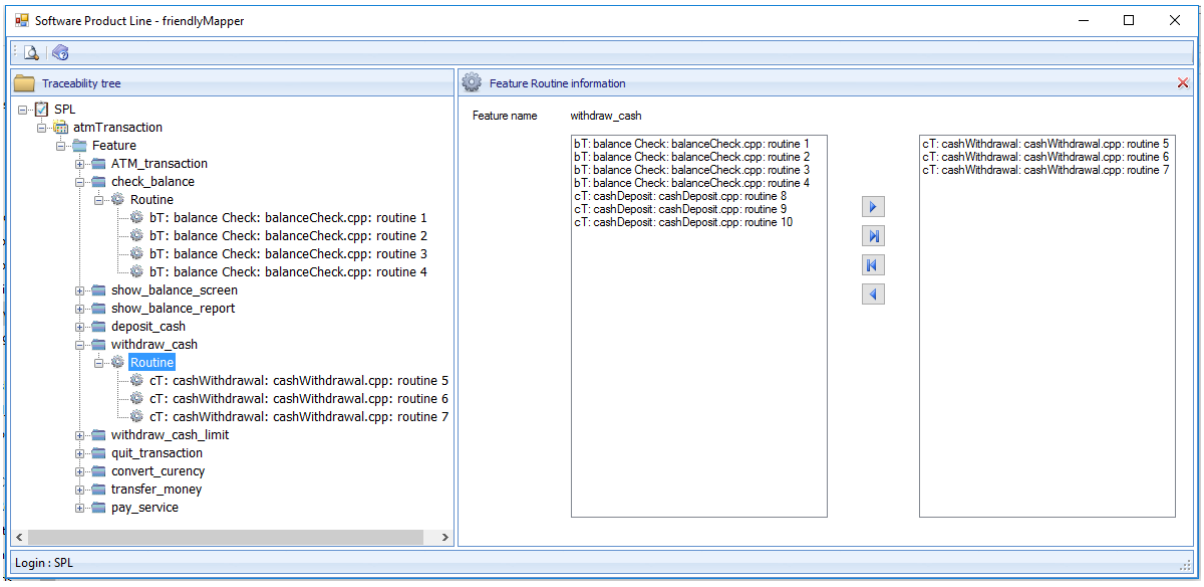


Figure 7.38: Traceability links of the ‘withdraw_cash’ feature.

‘deposit_cash’ feature to its related routines, using the tool, correspond to the traceability links appearing in row 5 of Fig. 7.20.

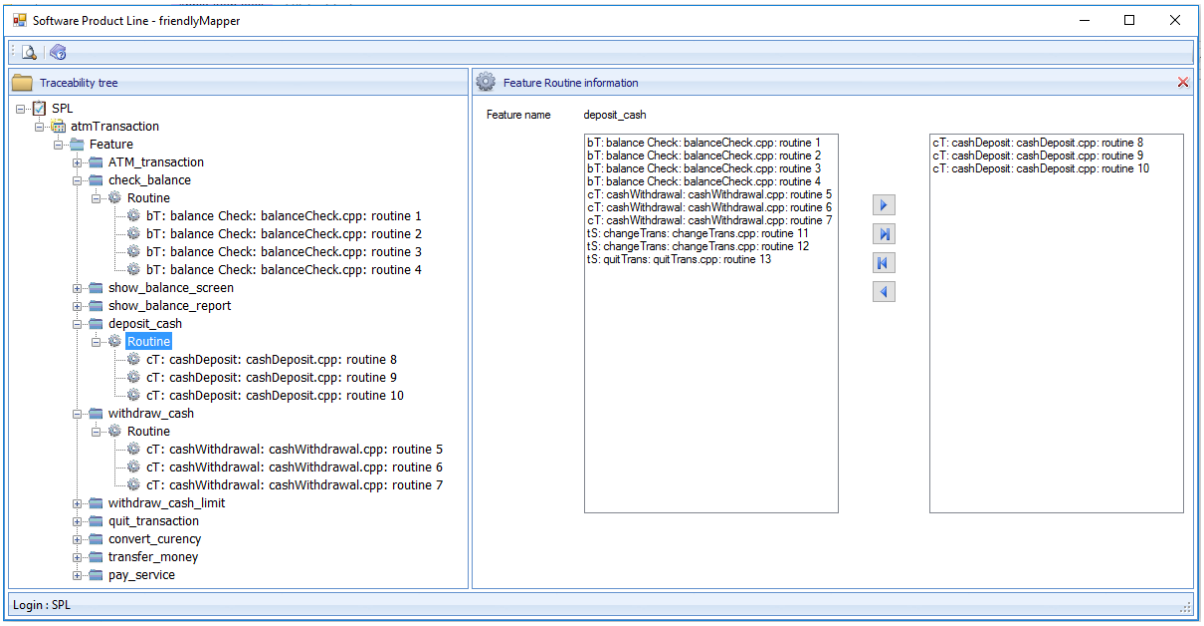


Figure 7.39: Traceability links of the 'deposit_cash' feature.

- Feature mapping conformance macro step: to support this macro step, the tool is able to perform the following.

  1. Update the features list of the traceability tree, by re-importing the features of the current FM from the XML file, which contains both the features of the ATM resulting SPL modelled by the current FM and new ones, as a result of refine the current FM with features of a new product . This involves adding the 'change_transaction' and 'show_message' features to the features list of the traceability tree. Fig. 7.31 (i.e., window 1) depicts the features list of the traceability tree that includes new features (written in a red color).
  2. Add routines of the 'change_transaction' and show_message' features to 'Routine list' list (see Fig. 7.40 and Fig. 7.41 – the routines are highlighted with blue color).
  3. Create traceability links from the 'change_transaction' feature to its related routines from 'Feature routine information' window (see Fig. 7.33). The traceability links from the feature to its related routines correspond to row 6 of Fig. 7.21.
  4. Create traceability links from the 'show_message' feature to its related routine from 'Feature routine information' window (see Fig. 7.42). The traceability links from the feature to its related routines correspond to row 7 of Fig. 7.21.
  5. Remove the 'show_message' feature and its related routine from the features list of the 'atmTransaction' traceability tree (see Fig. 7.43, Fig. 7.44, and Fig. 7.45). Fig. 7.43 shows the context menu that contains the 'delete' menu item of the

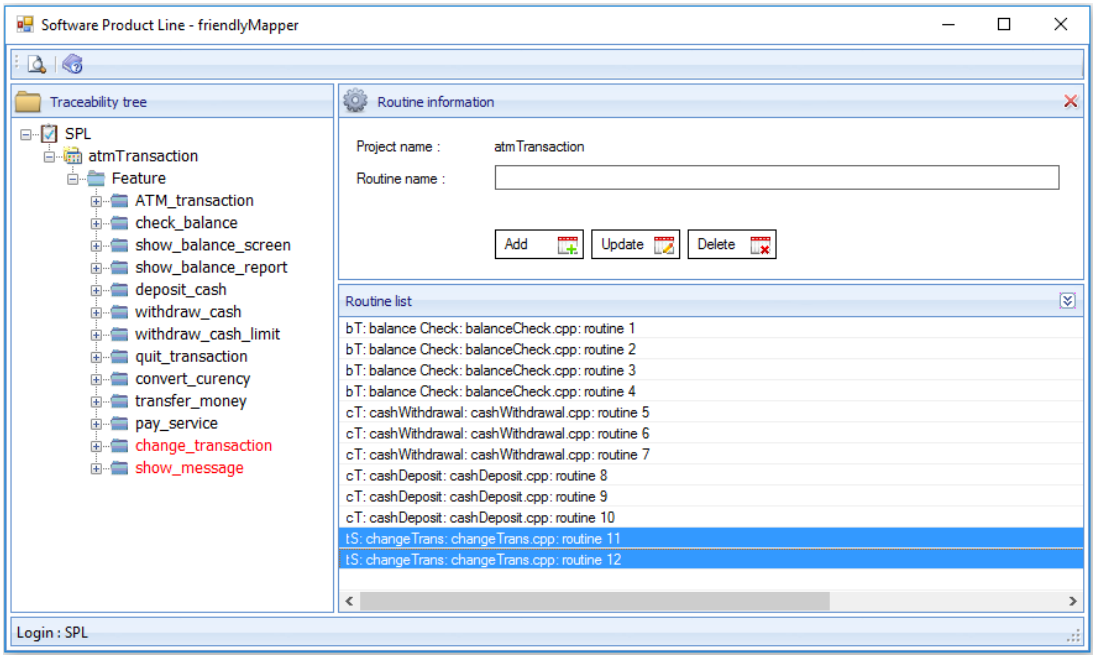


Figure 7.40: Routines of the 'change_transaction' feature.

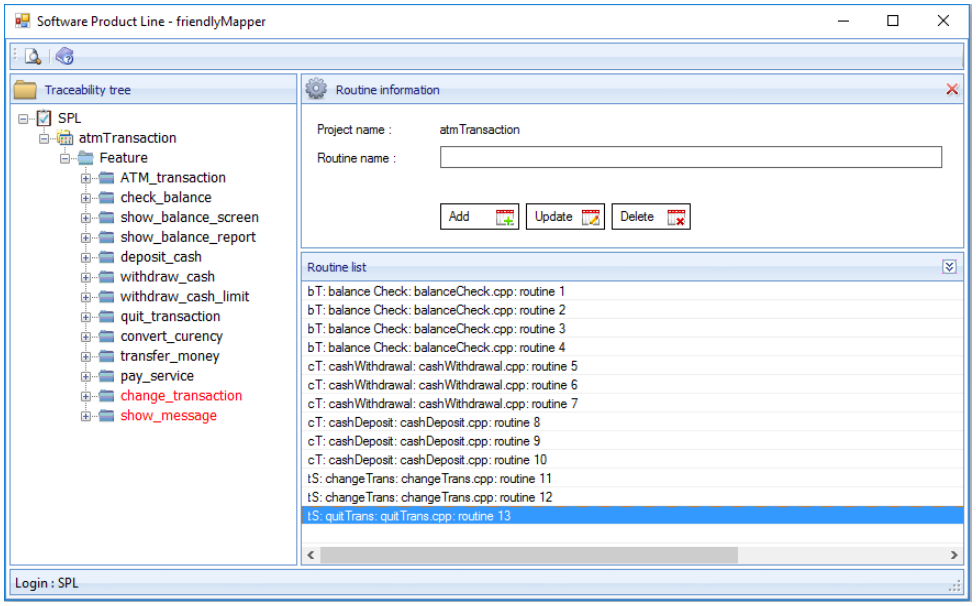


Figure 7.41: Routines of the 'show_message' feature.

'show_message' feature. Fig. 7.44 depicts the confirmation dialog that enables software engineers to confirm or revert the deletion process. Fig. 7.45 presents the traceability tree after the deletion task is achieved. Fig. 7.43 and Fig. 7.44 correspond to Fig. 7.22 appeared earlier in section 7.2, and Fig. 7.45 corresponds to Fig. 7.23 appeared in the same section.

6. Change the color of the 'change_transaction' feature from the red color to color

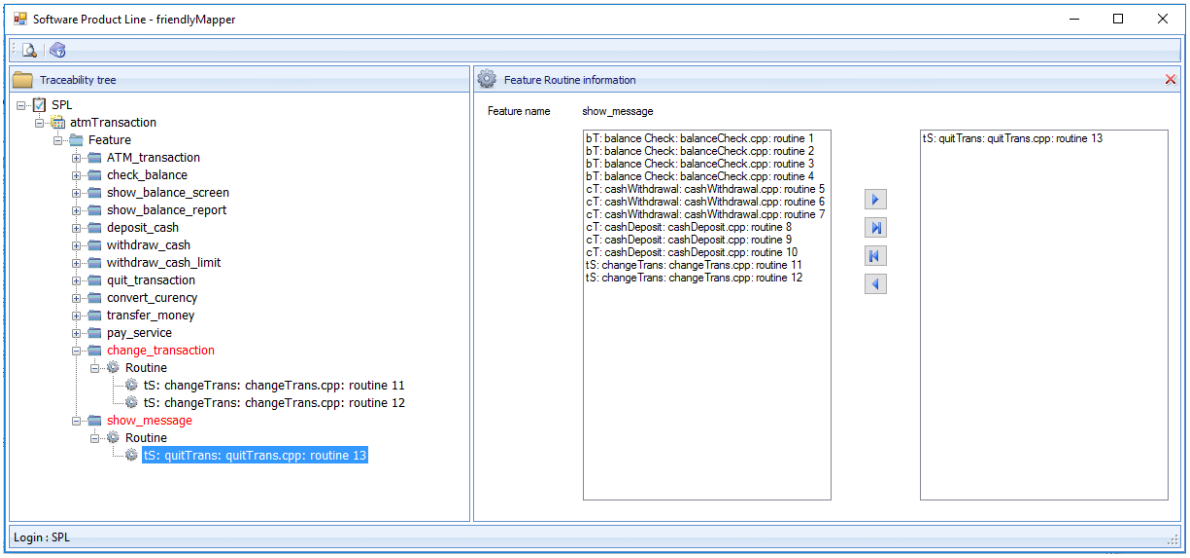


Figure 7.42: Traceability links from the 'show_message' feature to its related routines.

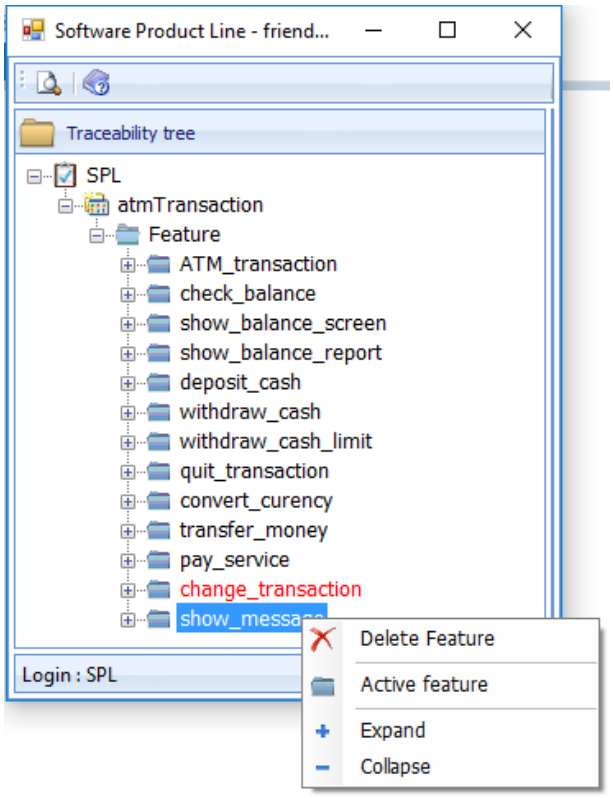


Figure 7.43: Delete the 'show_message' feature using 'Delete feature' menu item of the context menu.

of the remaining features of the traceability tree (see Fig. 7.46 and Fig. 7.47), by selecting the 'Active feature' menu item, upon right-clicking the feature .

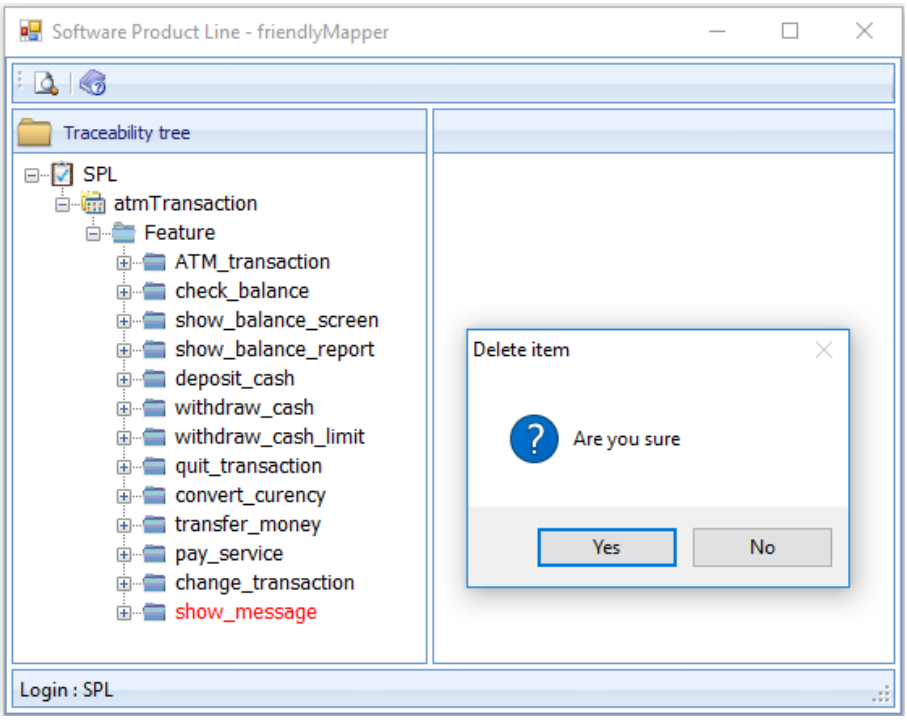


Figure 7.44: Confirm the deletion task.

7.3.3 Evaluation

We have evaluated the feature mapping activity and tool support as a part of evaluating our approach, by conducting a case study at Bosch Car Multimedia company. An existing classical sensor variants family, called CSVF, is used here for illustration purposes. We have used AUTOSAR architecture, which has been adopted by the classical sensor development team (CSDT) for a long time. Firstly, we have used the current FM, which has been derived during the first stages of the case study (see Fig. 8.6). Secondly, we have created traceability links in the traceability tree between the features and the feature-related-code fragments (i.e., routines). Thirdly, we have evolved the traceability tree by updating the traceability links to maintain the coherence whenever feature changes occur in the current FM. Finally, we have evaluated the performance of the tool.

Evaluation Objective The overall purpose of the evaluation is to validate that the steps of the feature mapping activity of the EvoSPL approach generally and the friendlyMapper tool specifically can be applied in the evolution of a realistic products family, to support feature-code mapping. Our evaluation is focused on showing that the capabilities of the tool are valid to be used to support traceability in the SPL context. First of all, we validate if each macro step of the feature mapping activity and its related steps are feasible. In particular, we are interested in whether or not they are compatible with a products family that has its own software architecture (i.e., reference architecture).

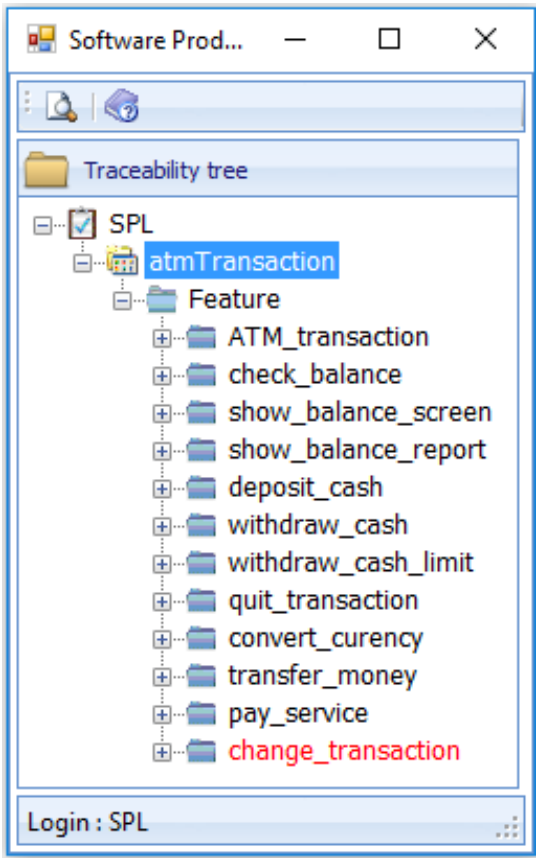


Figure 7.45: Features list of the traceability tree after deleting the 'show_message' feature.

We also have measured the number of the changes that need to be made to the code of an existing products family, to adopt our approach. We want to ensure that the tool and its underlying concept will not impose significant changes on a system that is developed using C-C++ programming techniques. Next, we have evaluated the capabilities of the tool including the creating and maintaining of the feature-code traceability links in the traceability tree. We have validated if the automatic creates, updates, and deletes operations of the traceability links, using the tool, works on the traceability tree correctly, and if the current FM and the traceability tree are consistent whenever a feature change occurs. Additionally, we have evaluated the performance of our approach (for comparison reasons, we called the feature mapping activity and the tool-related operations in this section as our approach) is acceptable compared with the normal approach.

Evaluation methodology We have evaluated our tool-based approach to an industrial C-based code of the CSVF. The variability in the code is implemented with the C pre-processor directives. The family has over 40 packages and more than 300 K SLOC. It has been through more than 3 years of development and has several major releases. The CSDT uses an ad-hoc reuse approach (i.e., clone-and-own approach), we called it here the normal approach.

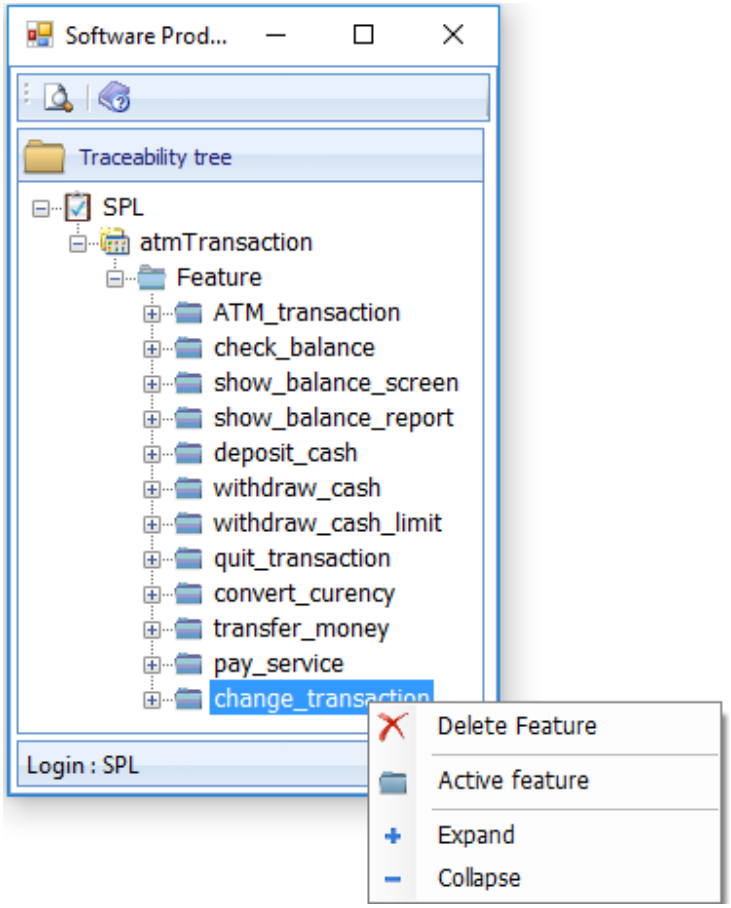


Figure 7.46: Change color of the 'change_transaction' feature using 'Active feature' menu item of the context menu.

The normal approach creates a new product branch based on the platform product. It reuses the platform product and modifies it to satisfy the new customer requirements. The products are given a specific name, like product and shall be identified by ascending number (e.g., creating product 111 where the last created product is product 110).

Many features have been added to the family, while the system evolved, such as 'layout', 'calibration'(or signa_12), 'fault_type' (or signal_4), and 'speed_validation' (or signal_9). In particular, many of these features and product-specific configuration settings can be enabled/disabled in the configuration files (i.e., header files) and using the pre-processor directives (i.e., #ifdef). This provides an opportunity for us to fully evaluate the approach's support for real different members of a products family. Besides, the CSVF has its build tool and the dependency embedded implicitly within the feature's implementation. In addition, the CSVF is initialized by a system header file based on the configuration file, and all the modifications are presented in the project (PRJ) package that is corresponding to the PRJ layer (the most upper layer of the

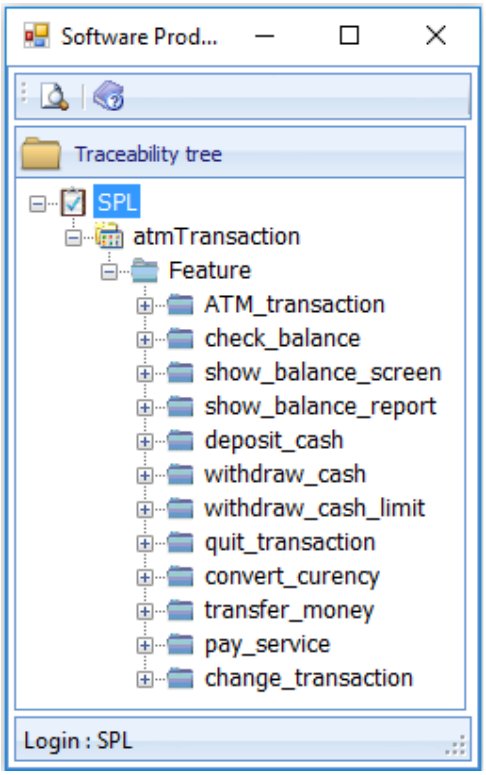


Figure 7.47: Features list of the traceability tree after changing the color of the 'change_transaction' feature.

AUTOSAR architecture) . All these properties have made CSVF appropriate for our evaluation given the objective described above.

The evaluation consists of the following three stages presented as follows.

Stage 1. Preparation and environment setup. The first task of the evaluation was to import the packages of the products related to the CSVF, including the code in the local development workspace of Eclipse IDE. One advantage of our approach mentioned earlier, it does not require any amount of changes to the code.

Stage 2. The reference architecture and code investigation, feature tree design (i.e., the current FM), and traceability links creation in the traceability tree. The ability to identify the features of the current FM in the code is called feature traceability [Mei+17]. We (1) have started first by investigating the reference architecture of the CSVF. The CSDT uses AUTOSAR architecture in expressing different types of variations, such as variations in the module's interfaces, variations at the architectural level, and so on. In addition, the relations between the variations are easily understood by the developers (members) of the CSDT.

As shown in Fig. 7.48, on the highest abstraction level, AUTOSAR has three software layers: Application layer (APP and PRJ), Runtime Environment layer (SL, ECUAL, MCAL, and CDD), and Basic Software layer, which run on a Microcontroller. The application layer is a specific part of AUTOSAR architecture; this part is presented in all products. At the same time, PRJ is a specific part of the application layer (every single product is built with its own PRJ layer).

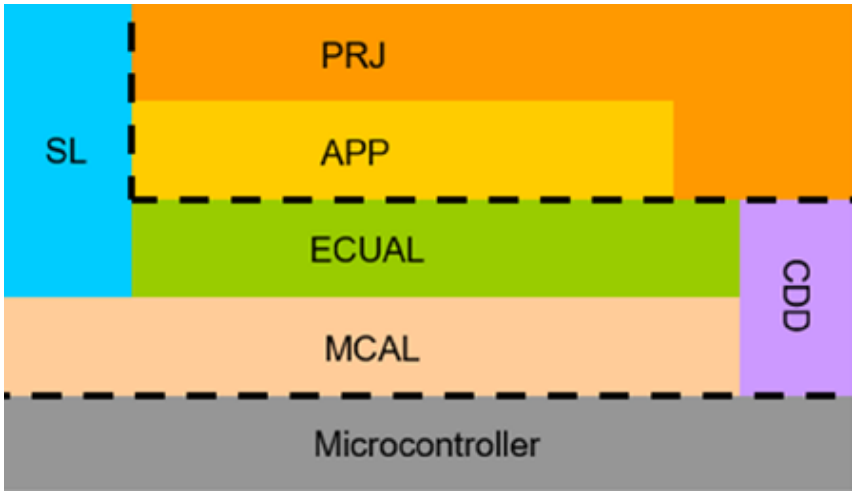


Figure 7.48: AUTOSAR layered architecture.

The PRJ layer is bound to the customer and sensor specifications and provides all project-specific configurations to other layers, where necessary [Gom13]. After that, we (2) have traced features of the current FM to feature-related-arch-elements (feature-related-subsystem and feature-related-component) of the PRJ layer of the AUTOSAR architecture (Feature- architecture mapping macro step). Additionally, we (3) have traced features of the current FM to feature-related-code-fragments, using the reference architecture as an intermediate artifact. Every subsystem and component in the PRJ layer correspond to a sub-package and module in the code, respectively. We have begun to identify, for each feature of the current FM, the feature-related-code fragment of the code, based on the variability information of the reference architecture.

Finally, we (4) have located the feature-related-code fragments, which are corresponding to a specific feature in the sub-packages, modules, and source files, including routines, statements, and expressions of the PRJ main package (folder). The feature identification was mainly based on (i) the configuration files, (ii) the RSDs, which represent the textual requirement in our case study, and (iii) the code comments (contain rich documentation that helps to specify feature-related-code fragments). Feature-related- code fragments were identified through semi-automatic analysis of the code assisted by Eclipse. We have analysed the variability facts in the pre-processor code. Then we have used Eclipse IDE to locate the feature-related-code fragments based on the following observations.

- The pre-processor code supports file inclusion (include), macros constant (define), and conditional compilation.
- In particular, many features can be enabled/disabled in the configuration file of the CSVF.
- Once the product is created, the product-specific configuration or modification has to be done using file inclusion. Normally, this inclusion contains a configuration and setting some values for the features. Since, we are interested in features and feature-related-code fragments, these files are irrelevant for the feature identification.
- The macro definitions and conditional compilation are the most-frequently-used features of the pre-processor. The CPP annotations have been used to delimit the optional code associated with each feature, such as #ifdef and #endif to surround the code fragments.
- The macro constants are defined with directive #define and are normally assigned values. Then the constants are used in #ifdef statements, which allow #ifdef blocks to be compiled conditionally into the corresponding code fragments. Since #ifdef block can be mapped to a variation, it is either optional or an alternative feature.
- Feature-related- code fragments are scattered over the multiple files and multiple routines.

After the features were identified, we have updated the traceability tree that was created earlier by either creating the new traceability links or updating the existing traceability links in the traceability tree whenever the feature changes occur. During this process, the traceability links were semi-automatically created and updated by the friendlyMapper tool.

Stage 3. Capabilities evaluation. Up to this point, we have the following CSVF artifacts: the reference architecture (i.e., AUTOSAR), the code consisting of core assets and feature-annotated code, the current FM, and traceability tree. Next, we have evolved the current FM designed using FeatureIDE, specifically, we have made a feature change, and we have used the friendlyMapper tool, to automatically update the traceability links in the traceability tree. Then, we have executed the updated traceability tree again. We have assessed the feature mapping and tool by (i) determining the feasibility of the approach, (ii) validating the coherence between the modified current FM and the traceability tree, (iii) inspecting the functional correctness of the updated traceability tree, and finally (iv) observing the performance of the tool.

Results In this title, we discuss the assessment of feature mapping activity, including friendlyMapper tool, as follows.

First. The feasibility of the approach was validated by the fact that we have successfully finished the creation process of the traceability links in the traceability tree, and all the functions of

CSVF are still working correctly in the final system. The reference architecture that we have used includes 56 components. There are 35 common components representing the basic software and hardware services application parts of the CSVF. These parts are also not included in the feature tree and consequently in the traceability tree as they exist in every instance. The application layer includes 60 features and 21 components. The PRJ layer inside the application layer contains the product specific requirements and has 4 components. In total, 300 feature-code relationships were created. This was determined by counting the number of traceability links in the traceability tree.

On average, each feature is related to five feature-related-code fragments (i.e., routines). All of them were automatically created in the traceability tree, using the tool, during the evaluation. We have compared the code base of the CSVF before and after the evaluation in terms of the numbers of modules, source files, routines, and lines of code (LOC), and we have found that our approach does not impose changes on the code of the CSVF. The only required changes are the ones mentioned in the traceability tree to guide software engineers to update the code to maintain its conformance with change in the requirements. This is the cost of developing and maintaining the CSVF whenever the feature changes occur.

Second & third. The feature-code coherence and the functional correctness of the updated CSVF was validated by the project manager. The CSDT have updated the code based on the traceability links of the traceability tree of the tool, upon feature change occurred in the current FM. The project manager of CSDT has confirmed that the reference architecture, the current FM, and code are consistent. We have asked them to automatically-update the traceability tree after each change (e.g., adding/removing a feature) and to manually-update the code to maintain its conformance with the current FM. This allowed us to verify the functional correctness of the updated CSVF, using our tool-based approach.

As an example, the consistency of the traceability tree is verified, by adding a new feature to the current FM first, after that, we have automatically updated the traceability links of the traceability tree (i.e., we added new traceability links from the new feature to routines that implement this feature). For instance, the CSVF can write sensor commands in a message in several formats. We have added a feature to represent the new message format (i.e., identification), then we have asked the software developer of the CSDT to update the code of CSVF, based on retrieving the traceability links for the new feature from the traceability tree. Fig. 7.49 presents the features of the CSVF and exemplifies (in the right-hand side) the traceability links between the ‘identification’ new feature (presented in a red color) and its related routines. For readability’s sake in Fig. 7.49, the features list of the traceability tree is presented in two screen-shots (left-hand side and right-hand side of Fig. 7.49). Also, ‘signal_4’ and ‘interface support’

features represent changes of the current FM (i.e., new features) ; thus, the friendly mapper automatically presented them in a red color, when it has re-imported the features from the XML file to the traceability tree.

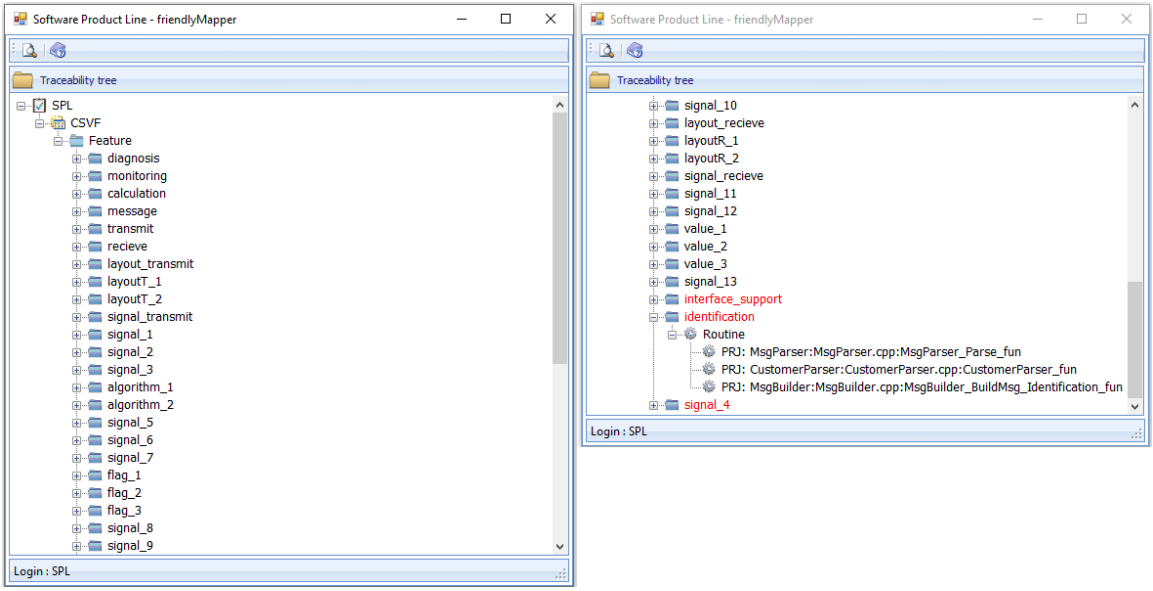


Figure 7.49: Features of the current FM and the traceability links between the 'identification' feature and the corresponding routines.

If the new message type (i.e., identification) is added to the current FM of the CSVF, the corresponding routines should be enabled to activate this feature. The software developers have retrieved the traceability links of the 'identification' feature, and then they have updated the code to maintain the new functionality. As a result, we have used friendlyMapper to add the new feature to the traceability tree and to assign the new feature to its related routines. With the new feature completely added to the CSVF artifacts (i.e., the current FM and code), we have noticed that the updated system replied to the request for reading the new feature from the XML file representing the current FM. Respecting the confidentiality issues related to the company, we hide name of signals for the CSVF. The tool is available online in [Fri].

Fourth. Performance evaluation. In terms of performance, all the major operations (e.g., create, update, and remove traceability links) that we have evaluated have finished rapidly (e.g., in several seconds). This is a benefit derived from automating the creation and updating of the traceability links in the traceability tree. To measure our tool-based process that has been proposed to improve software developer's performance. Instead of the normal approach that the company has been adopted. Therefore, we have defined the dependent variable 'efficiency', which is calculated as the ratio between the numbers of right updates of the code according to feature changes (updates scenarios) and the total time spent on performing the updates scenarios. To

carry out the experiment, we have asked ten software developers (participants/members) of the CSDT to perform two trials, in which they had to achieve tasks related (1) to add new features to the current FM, and then (2) to update the code to maintain its coherence with new requirements.

In one of the trials, all the participants have used the normal approach to perform the tasks. In the other trial, the same participants have used our tool-based process to perform the same tasks. At the end of the two trials, the 'efficiency' of each participant performed was calculated. To determine whether the mean difference of the dependent variable (i.e., efficiency) is considered to be extremely statistically significant in participants performance between the two approaches (i.e., when using the normal approach compared to our tool-based approach), a paired t-test was used with the following hypotheses.

- Ht0: there was no change in the performance of a software developer using our tool-based approach.
- Ht1: there was a change in the performance of a software developer using our tool-based approach.

Table 7.9 shows the result of the paired t-test related to comparing the average of the 'efficiency' for the software developers using the normal approach and using our tool-based approach. The output provides useful descriptive statistics for the two approaches that we have compared, including mean and standard deviation, as well as actual results from the paired t-test. Looking at the mean, we can see the participant who have used our tool-based approach had a higher 'efficiency' at the end of the experiment compared to those who have used the normal approach.

We can see that there is a mean difference between the two trials of 0.069 with a standard deviation of 0.0404, where df[1]=9. We have obtained the t-value of 5.4008, which is greater than t-critical value, and the p-value of 0.0004. By conventional criteria, this difference is considered to be extremely statistically significant. As the t-value is greater than the t-critical value and p-value is less than the threshold (i.e., $p < .05$). This eventually led to the rejection of the null hypothesis Ht0 and the acceptance of the alternative hypothesis Ht1. In other words, one can conclude that the performance of software developers has been improved after using our tool-based approach.

Moreover, the result reveals that the standard deviation of our tool-based approach (0.0360) is less than the standard deviation of the normal approach (0.0560). This leads to the conclusion

[1]DF (Degrees of freedom) is related to your sample size and shows how many 'free' data points are available in your test for making comparisons.

Table 7.9: Analysis of the objective dependent variable.

df=9	our approach	normal approach	
mean	0.2660	0.1970	
standard deviation	0.0360	0.0560	
paried t-test results of the experiment			
t-value	5.4008	t-critical value	2.2621
p-value	0.0004	Sd err	0.0130

that there may be a larger gap between one data value and another in the normal approach and the data values are all close together in our tool-based approach. As we have used the software developer from different experience/skills and background level, we can conclude that our tool-based approach helps software developer (1) to achieve convergent performance regardless of their skills and (2) to avoid difficulties within identifying the SPL evolution scenario.

Threats to Validity There are several validity threats to the evaluation of our tool-based approach. First, the study is limited to a single products family (i.e., CSVF), and also it is limited to a single development team (i.e., CSDT). To compensate for this limitation, we plan to go on evaluating our approach in other companies. Second, during data collection, we have mostly used a single person (the project manager) to evaluate and review the correctness of our approach and noting its artifacts. Also, we have only used ten software developers to join the evaluation. In extending this work we should, of course, include more than one products family and more software developers with different levels of working experience.

Third, regarding the estimation of the efficiency of our approach and the comparison with the normal approach, the software developers have made first the tasks with our approach and, later, they have made it with the normal approach they have used. Repeating a task may affect the time required to execute them. If both tasks are similar, the second task may be done in less time than the required, if they make the tasks without previous experience. Fourth, may the members of CSDT adopt the same viewpoints regarding our approach and we loss of information. This threat was mitigated by separating the interviews in two sessions and performing each interview separately. The first session was conducted and documented at the beginning of the month, and the second session was conducted and documented at the end of the month. Finally, threats to validity compromise our confidence in stating whether the study's result is applicable to other products family, SPLs domains, development teams, and companies.

Chapter 8

Evaluation

This chapter presents the evaluation of the EvoSPL approach conducted in this thesis. The evaluation was enacted in a coordination with the hypotheses introduced in chapter 1 of this thesis. The first section introduces the evaluation concept, the second section describes and explains the selected case study that we have conducted, to validate our approach, and the final sections provides the evaluation, obtained results, and threats to validity.

8.1 Evaluation overview

The EvoSPL approach was applied in an industrial-sized case study, in the automotive domain. In specific, the case study was conducted at Bosch Car Multimedia company, using the CSVF (see CSVF description below). The members of the CSVF are fully integrated in one platform. The implementation language used is C. To validate the contributions of this thesis, empirical study, surveys, interviews, and direct observations of the members of CSDT were performed. This puts the applicability of the approach, overall process, and results in practice.

Evaluation concept In order to evaluate the Evo-SPL approach, we have conducted a case study following the guidelines presented in [Lin+15] [KP98]. According to [Lin+15], the case study is composed of five major process steps to be walked through: planning, design, data collection, analysis, and reporting. We apply our approach on an (industrial) real case study: the CSVF. The case study is C-based software systems. We executed all the evaluation steps on the Windows 7 system, running at 2.4 GHz with 4 GB of RAM.

The selected case study is used to assess the phases of our approach in the field of our study. The advantage of having an industrial case study that has medium-sized systems is that such systems implement variability at different levels (package, module, routine, routine body, and attribute). Furthermore, the industrial case study allows us to put our approach in an environment of practice for the industry, which nominates it to be a successful example to adopt SPLs in

medium-sized companies of the automotive domain [IF19]. We used five products for the CSVF. The selected products (i.e., number of products) are covering all known features of the family.

Fig. 8.1 presents the structure of the evaluation concept. Evaluation questions were defined for the main hypothesis and empirical study was identified to answer the evaluation questions. Furthermore, empirical study was undertaken to prove the hypotheses from 1-6 (see Fig. 1.2), where we have found the empirical study including, survey, interview, and observation are applicable to the environment of our case study. Details about the evaluation are presented in the according sections of this chapter.

Evaluation questions The following evaluation questions are derived for the main hypothesis.

Evaluation question 1. Does the EvoSPL approach derive the current FM that models the CSVF successfully, including, almost, all known features of the family?

Evaluation question 2. Is the EvoSPL approach able to refine the current FM with features of (new) products successfully?

Evaluation question 3. Is the EvoSPL approach able to map features of the current FM to feature-related-code fragments of the code (both artifacts are related to CSVF) successfully?

The evaluation process and the empirical study defines the 'successfully' expression, as a positive feedback from CSDT and project manager that the EvoSPL approach is effective, efficient, and correct, in performing the proposed tasks, while validating the hypotheses and answering the evaluation questions. Fig. 8.1 depicts the evaluation structure and describes the relationships among the hypotheses, evaluation questions, and the evaluation criteria and methods.

The evaluation questions stated above were answered with different methods. Fig. 8.1 provides an overview about the evaluation criteria and evaluation methods. Besides, it determines which evaluation criteria was targeted with whom (CSDT or PM – project manager).). Here, 'industrial case study' means that (1) the EvoSPL approach was applied to the selected case study in an industrial company of the automotive domain, and (2) the evaluation questions were answered based on evaluation methods captured within the case study, in cooperation with the project manager and CSDT. Interview and survey mean that the evaluation question is answered based on the answers given by CSDT.

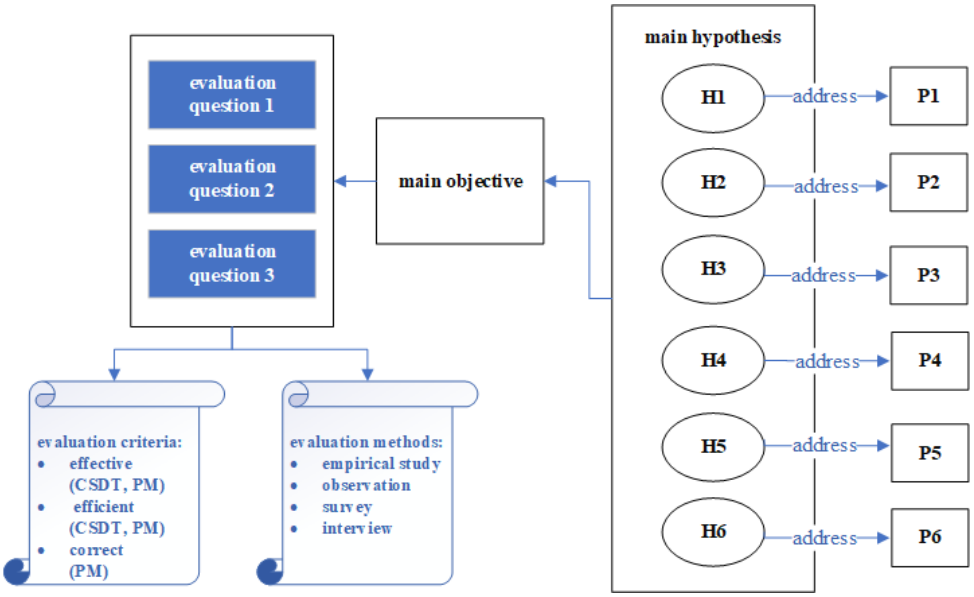


Figure 8.1: Evaluation structure.

8.2 The industrial case study

This section presents the case study that was conducted at Bosch Car Multimedia company. It is considered as an industrial-sized case study consisting of four stages. The concept of the evaluation process is inspired by the work explained by Linaker et al. [Lin+15]. We should mention here that the thesis is an industrial research describing an approach that handled SPLs evolution and variability in the domain of the automotive systems, to satisfy a range of requirements, by automotive industrial partners. Moreover, the thesis delves into how variability can be managed at requirement-level and how traceability of variability models (i.e., the current FM) to reference architecture and implementation code can be constructed. The industrial-sized case study was used to evaluate the approach, and it provided initial insights on its usefulness.

8.2.1 The products family used in the study

The story Over the past few decades, many industrial companies, like Bosch, have transitioned from single-systems development to products family development, in order to increase software customization, shorten the time-to-market, and improve the quality of products. However, there are many challenges that companies face with respect to adopting an SPL approach instead of ad-hoc reuse and in modelling and managing the variations in their product families. For instance, the need for systematic methods, to identify and exploit variabilities, appropriate modelling methods that are able to represent thousands of variations, efficient methods to detect and resolve dependency interactions among variations at different abstraction levels, and techniques to trace variability information among development artifacts.

The CSDT at Bosch Car Multimedia company explicitly adapted an ad-hoc reuse (i.e., branching). The team adopts the AUTOSAR architecture and reuses requirements documents, and code from earlier, by copying them from the artifacts of the platform (or initial release product). This is because most of the products are based on platform product. When adopting the ad-hoc reuse approach, it is often to extract an SPL of products family from the existing products, which is called reverse engineering [AM14].

CSVF description The CSVF is an industrial C-based code, which is widely used for designing and implementing sensors of the steering wheels braking systems of the different vehicle models [Bos]. The variability of the code is implemented with the C pre-processor directives. The CSVF has been developed and customized by the team for more than 3 years, to satisfy the needs of different customers in the automotive domain. Moreover, it has around 20 major releases and includes 50 packages. A number of features have been added and modified, while the product variants have been evolved over time [Bos].

To conduct this case study, we have used the CSVF, which has the AUTOSAR reference architecture. We have selected a group set that has five products of the family, based on consultation of the project manager. Naturally, the selected products cover all features of the family. The products of the CSVF, which were developed according to the needs of customers in the automotive domain, and a new product, which was an upcoming product scheduled for being developed in the near future, upon receiving a new customer request, are presented in Table 8.1. All the products were cloned from the platform product (or initial release product, which is often evolved from the platform developed and successfully used by the first customer) and then modified according to customer needs. To simplify the following discussion, we have assigned a number for each product of the CSVF and called the upcoming product 'Product New'.

Variability of the products from the automotive customer perspective The products of the CSVF[1] presented in Table 8.1 covers almost all the functional features (i.e., functional /customer perspective) of the vehicle models supported by the company. Each of these vehicle models has its own configuration of mandatory and optional features, while sharing a few common features with other models of the same vehicle type. To increase product quality and reduce cost and time-to-market, it is in the best interest of the automotive manufacturer to reuse existing product artifacts to build vehicle products in a systematic way, using SPLs development.

The main goal of our study was to find out how the EvoSPL approach initiates the resulting SPL of the CSVF from the individual members of the family and evolves the resulting SPL whenever a new product request is received. To investigate the development environment, we

[1]For confidentiality, not all details of the CSVF are provided (e.g., code).

Table 8.1: CSVF products .

Product no.	Product description	Package
Product 1	The optional layoutT_1, layoutR_1, algorithm_1, and flag_1 features enabled.	Product1CSVF
Product 2	The optional layoutT_2, layoutR_2, algprithm_2, signal_4, flag_2, and signal_ 9 enabled.	Product2CSVF
Product 3	The optional calculation, layoutT_2, layoutR_2, algorithm_2 and flag_2 signal _9, signal_10, value_3, and signal_13 features enabled	Product3CSVF
Product 4	The optional identification, layoutT_1, layoutR_1, algorithm_1 and flag_1, signal_10, and value_3 features enabled.	Product4CSVF
ProductNew	All optional features enabled.	Product5CSVF

have conducted interviews with the key person (i.e., the project manager) that has been involved in the CSVF for quite some time, and one of the members who has more than three years of development experience. Furthermore, we have studied requirements documents (i.e., RSD presented in the development process as CAN Matrix), design documents, and the reference architecture (i.e., AUTOSAR architecture) that we got from the company.

Fig. 8.2 depicts the current FM of the CSVF resulting SPL as manually defined by the project manager and CSDT and designed by us, during the first interview. It shows 36 functional requirements (features) and the 'CSVF_SPL' root feature. Some features are common, such as 'diagnosis', 'monitoring' and 'calculations' and some features are optional, such as 'calculations' and 'interface_support', both groups have no child features. The other feature 'message' represents the main functional concerns of the CSVF. It includes two main group features: 'transmit' and 'receive'. Each group consists of several features that specify its layout and signals that perform the main functionalities of the sensor. Some of them are common (e.g., signal_1) and some of them as optional (e.g., signal_9). For example, the 'transmit' feature has two layouts: 'layoutT_1' and 'layoutT_2' and has12 child-features (signal_1 to signal_10 and other features). The 'receive' feature also has two layouts: 'layoutR_1' and 'layoutR_2' and other has 3 child-features (signal_11 to signal_13). Some features form an 'xor-group', such as 'flag_1 , 'flag_2' and 'flag_3', and 'value_1', 'value_2', and 'value_3'.

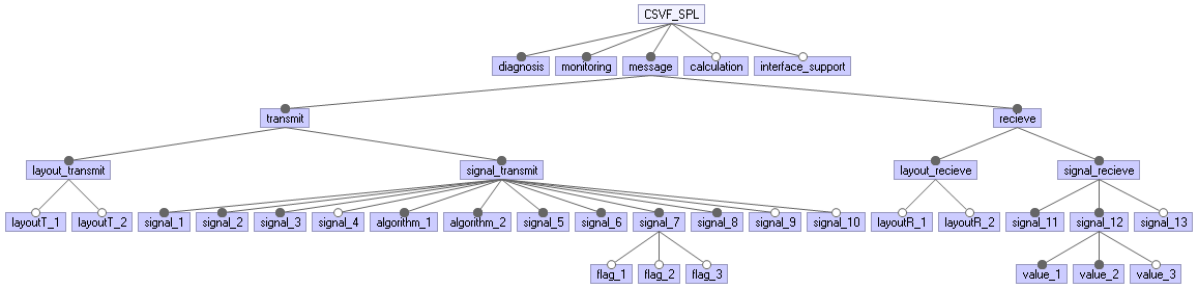


Figure 8.2: The current feature model for the classical sensor variants family resulting SPL defined by the project manager.

CAN Matrix For technical issues related to sensors, the requirement documents of the CSVF are written in an RSD format and called CAN Matrix [Bos]. The CAN Matrix consists of sections, which are visually separated from each other with a section break, typically consisting of extra space between the sections, and sometimes also by a section heading for the latter section. The basic sections of the CAN matrix are presented as follows.

1. The CAN parameters section specifies the technical values related to the CAN Matrix.

2. The messages overview section clarifies the message kind. The CAN Matrix supports three kinds of messages; transmit , identification, and receive message. The transmit and receive message are common among the CSVF, but the identification message is varying among the family. The transmit message (and receive message) and identification message corresponds to Common feature and Optional feature in the current FM respectively (see Fig. 8.2)

3. The standard message section lists and explains the signals of the standard message (called also transmit message), such as angle, speed, counter, and calibration, TRIM, and flag. The layout of the standard message is an important feature that may vary among the CSVF. The variation regards layout name and number of bits.

4. The configurations message section specifies the signals that are used to calibrate the sensor. The layout of this message varies among the CSVF. This message (called also receive message) includes important signals, such as calibration angle and calibration command. The calibration command signal calibrates the steering angle sensor by setting the angle signal to a specific calibration angle value. Some calibration commands are common, and others are variable. For example, the steering angle sensor can be re-calibrated (optional feature ‘value_3’) without resetting the calibration. This depends strongly on both vehicle model and manufacturing settings.

Finally, it is worth mentioning that the CAN Matrix contains written notes. The notes address the issues related to signals and specify the constraints and dependencies among them. While different CAN Matrix documents usually differ from each other, e.g., in the supported message layout, but their underlying sections are very common, as well as can be written from reusable documents. Hence, a company offering a variety of product variants (and the related CAN Matrix) can adopt an SPL approach to achieve a systematic reuse. The variability between potential product variants (e.g., CAN Matrix documents) can be captured in a variability model (e.g., an FM). It defines the available product variants, e.g., different signals, message types, and layouts support for the CSVF.

There are several variability models that could be applied here. In this case study and to follow the EvoSPL approach, we use FMs (i.e., the current FM) as a basis for modelling variability. A mapping is then specified where each feature of the current FM corresponds to one or more code base artifacts (i.e., module, source file, and routine).

CAN Matric and the current FM A feature in the CAN Matrix is a keyword that has a semantic and relevant concept to functionalities of sensor or automotive customers, such as signals, technologies, services, and actions. The current FM, shown in Fig. 8.2, clarifies that each product must support the standard message (transmit message) and configuration message (receive message). Both are common/mandatory features, and we have found them documented in all the CAN Matrix artifacts of the CSVF. A product may include the 'identification' message (optional feature) and must include layout feature (common/mandatory feature) from which one child feature has to be chosen ('xor-group' relation). Since, the 'identification' feature is optional, it is documented only in the CAN Matrix artifact of some products (i.e., Product 4 and ProductNew).

Code and variability realization of the CSVF The research investigation of the code that belongs to the CSVF reveals that (i) the C files are the type of solution artifact used by the team (i.e., CSDT), to develop the individual products of the Family. Although the research indicates that (ii) several mechanisms are being used to realize variability in the code of the CSVF, but #ifdef blocks and if-statements are the most commonly variability realization mechanism used by CSDT. The team prefers to limit their use of # ifdef blocks for various reasons. For instance, they see that too many #ifdef blocks affect code readability. In some cases, alternatives to #ifdef blocks are used such as by deferring the binding time to runtime in configuration files (#ifdef blocks supporting static binding time only) or by adding #includes. For that, the second most common variability realization mechanism used by CSDT in the code implementations is configuration files.

Table 8.2: Our approach evaluation Steps.

Step 1	Design of the case study (planning).
Step 2	Preparation for data collection.
Step 3	Execution of the data collection on the case study.
Step 4	Analysis of the collected data and reporting.

Regarding the normal ad-hoc reuse approach adopted by CSDT, originally the platform product is used and cloned into a second product, which then adapted and evolved on its own to satisfy customer needs and successfully was used by the first customer (initial release product). The cloning (branching) continued for several other products, simply by branching with the version control system, since all the products are related to the integrated platform. The developers used Beyond Compare tool, to compare the first two products, merged their commonalities to form a unified code base and realize their variabilities, by introducing variations using the preprocessor directives. This process continued for the other products; however, it was challenging to reach a full integration for the reasons related to limitation in Beyond Compare as described by the interviewees.

8.3 Evaluation of EvoSPL approach

To evaluate the EvoSPL approach using the artifacts of the CSVF and incorporation with CSDT, we have prepared and performed an empirical case study that includes different evaluation methods (empirical study, observations, survey, and interviews), to answer the evaluation question stated in section 8.1. As illustrated in Table 8.2, the empirical study consists of four steps presented as follows.

8.3.1 Design of the case study (Planning)

Good planning is necessary for the success of the case study. Therefore, we planned several issues.

1. Objective. The objective of the case study is to prove the fitness of the EvoSPL approach for the automotive domain, using the CSVF. We have planned the case study to be conducted at Bosch Car Multimedia company, with CSDT, who has previous experience in the automotive domain development.

2. Treatment. Our case study has one treatment, which is the EvoSPL approach. We have planned to apply the process of the approach to artifacts of the CSVF, using the same setting and daily working environment of the team.

3. Objects. The object of our case study is the CSVF implemented based on AUTOSAR architecture, and its related artifacts including the CAN Matrix, reference architecture, and code.

4. Subject. The subjects of the study are the individual members of CSDT and the project manager. To prove the 'usefulness' of our approach, we have chosen the members of the team with a different level of skills and experience.

5. Method. We have planned to perform our case study in two stages. In the first stage, we have taken the role of the software engineers and we have applied the EvoSPL approach on artifacts of the CSVF. During the execution of the EvoSPL approach process, initially, we have derived the FL and the current FM. After that, we have started bootstrapping the products of the CSVF to the initial SPL, to get the bootstrapped SPL, and then we have evolved the bootstrapped SPL with a new product, using the feature model refactoring scenario, to get the resulting SPL. Finally, we have mapped each feature to the corresponding feature-related-code fragments.

   In the second stage, we have planned to evaluate the EvoSPL approach and the generated artifacts, as a result of applying our approach, using several data collection methods, where we have found the empirical study including surveys, interviews, and observations are applicable to the environment of our case study (see Appendix B). We have arranged (1) to invite CSDT to join a training session, (2) to apply an empirical study that includes an empirical evaluation and a survey questionnaire, and finally,(3) to attend individual interview sessions.

8.3.2 Preparation for data collection

We have prepared many documents suited to the collecting data methods used in the empirical study. We have designed a PowerPoint Presentation that introduces the EvoSPL approach to CSDT, in addition, we have prepared a survey that is a part of an empirical study. The survey consists of 22 questions. These questions were formulated by using a combination of descriptive, behaviour, and attitudinal questions. The answers were written using ordinal and nominal scale response format. One of the most important members of CSDT is the project manager, for that, it is of special importance dedicating her a survey that contains measurement items related directly to our approach hypotheses.

For the empirical study, we have defined a set of items to be evaluated, by asking CSDT to perform specific tasks and then answer questions while interacting directly with our approach. Ideally, to compare the developer's solutions with the correct solutions, in order to investigate our hypotheses, we have already defined the correct solutions for the tasks.

Case Study environment and procedure The study was conducted in the software development department, during October 2017- October 2018 at Bosch Car Multimedia company. The study was conducted in two stages. In stage 1, we have taken the role of software engineers and we have applied the EvoSPL approach on the CSVF, starting with requirement documents (i.e., CAN Matrix) of two products of the family. In stage 2, we have evaluated the 'effectiveness' and the 'efficiency' of the EvoSPL approach from the point of view of CSDT and we have evaluated the 'correctness' of the EvoSPL approach concerning project manager perspective.

For the direct methods to collect data, we have prepared semi-structured interviews with the members of CSDT, to get a direct feed-back related to the EvoSPL approach. The interview dialog was guided by a set of questions (i.e., open questions). Moreover, in order to get a deeper understanding, we have informed the project manager (and she has informed the team via a formal email and meeting) that we were investigating how the members of CSDT used the artifacts of our approach in their normal daily work. We have agreed with them to use the artifacts of our approach daily for three hour. We have taken notes about the observations.

Summary of the generated artifacts Following the EvoSPL approach activities: difference analysis, variability analysis, feature model synthesis, bootstrapping, evolution, and feature mapping, during the execution of the case study, stage 1 were performed as goes after.

- Firstly, the difference analysis and variability analysis steps were applied, and the SMD and FL of the CSVF were defined, capturing commonality and variability between Product 1 and Product 3 (see R6 of specific requirements of the difference analysis activity in chapter 5), using the CAN Matrix of each product. The FL of the CSVF contains a list of features and a reference for the ARs that specify each feature. We have selected one of the products (Product 1) that match the initial release product and another one (Product 3) that has the most functionalities (features) among all the CSVF.

- Secondly, the current FM was derived, using the steps of the feature model synthesis activity, representing the CSVF initial SPL, at a high level of abstraction. The model, shown in Fig. 8.3, presents the common features, such as 'diagnosis', 'monitoring', 'message', and the optional one 'calculation'. The 'transmit' feature has two features, involving 'layout_transmit' and 'signals_transmit', where the 'layout_transmit' feature has two alternatives, which are 'layoutT_1' and 'layoutT_2'. This means that the family supports two different message layouts, one for each product: the first layout is 'layoutT_1', which has five bits and the second one is 'layoutT_2', which has seven bits.

  Concerning the 'signals_transmit' and 'signals_receiv' features, each of them consists of different signals for the products (Product 1 and Product 3). Some of them are common,

like ‘signal_1’, ‘signal_2’, ‘signal_3’, ‘signal_5’, ‘signal_6’, ‘signal_7’, ‘signal_8’, ‘signal_11’ and ‘signal_12’, as well as some of them are optional, ‘signal_4’, ‘signal_9’, ‘signal_10’, and ‘signal_13’, and some of them are within alternative group, like algorithm_1 and algorithm_2 as well as flag_1and flag_2. Finally, ‘signal_12’ is a group feature that consists of value_1 and value_2 common features, and one optional ‘value_3’ feature.

- Thirdly, after constructing the current FM, it is the time to perform the bootstrapping and evolution activities. The former involves bootstrapping the remaining products (Product 2 and Product 4) of the family to the CSVF initial SPL, to deliver the CSVF bootstrapped SPL. The latter entails encompassing the bootstrapped SPL with a new Product (ProductNew), to come up with the CSVF resulting SPL. Both activities use the feature model scenario steps to evolve the current FM with requirements (features) of the (new) products.

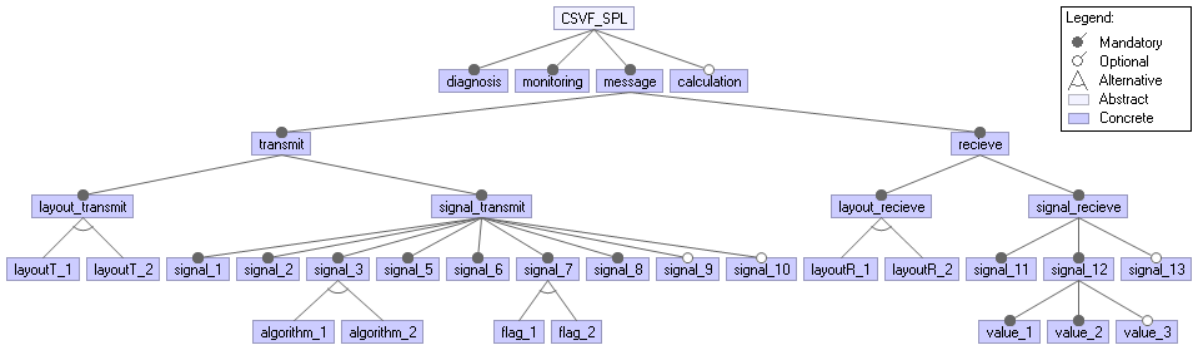


Figure 8.3: The derived current FM from Table 8.1.

We have identified and stored the features of Product 2 and Product 4 in a FL of each product. At this point, we can refine the current FM with features of those products, using the feature model refactoring scenario steps. Now the SPL is bootstrapped completely and the commonality and the variability of the CSVF were presented by the current FM, at refactoring point1 and refactoring point 2 (see Fig. 8.4 and Fig. 8.5).

The current FM at refactoring point 1 Fig. 8.4 depicts the current FM at refactoring point 1, which includes refining the derived current FM shown in Fig. 8.3 with features of Product 2. We have applied the feature model refactoring scenario on the derived current FM and the FL of Product 2. The refactoring process results in adding a new optional ‘signal_4’ feature to the current FM (see Table 6.2). The feature exists in the FL of Product 2 and does not exist in the derived current FM, which leads to applying the proper refactoring notation (see A catalog of sound FM refactoring in chapter 6) for the current FM (i.e., Refactoring 12. Add Optional node).

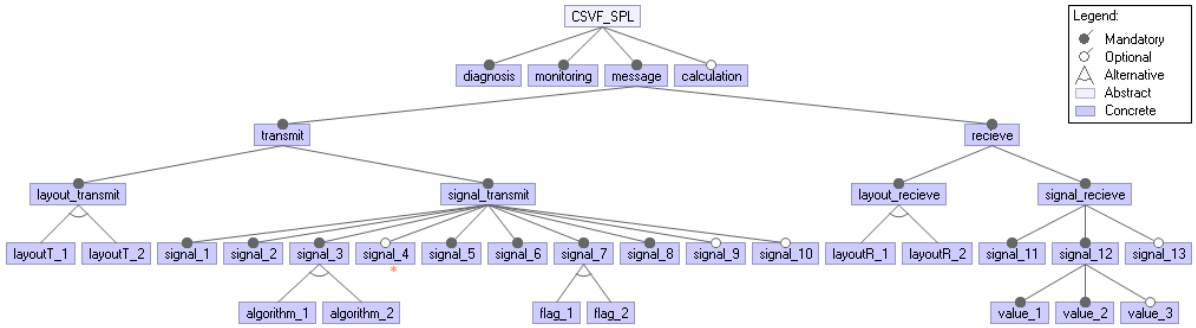


Figure 8.4: The current FM at refactoring point 1 from Table 8.1.

The current FM at refactoring point 2 Fig. 8.5 depicts the current FM at refactoring point 2, which includes refining the derived current FM shown in Fig. 8.4 with features of Product 4. We have applied the feature model refactoring scenario on the derived current FM and the FL of Product 4. The refactoring process results in adding a new optional 'identification' feature to the current FM (see Table 6.2). The feature exists in the FL of Product 4 and does not exist in the current FM at refactoring point 1 (Fig. 8.4), which leads to applying the proper refactoring notation (see A catalog of sound FM refactoring in chapter 6) for the current FM (i.e., Refactoring 12. Add Optional node).

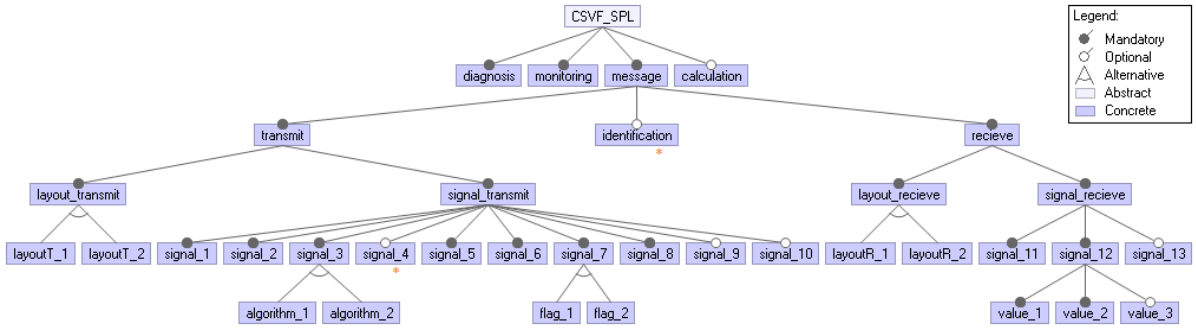


Figure 8.5: The current FM at refactoring point 2 from Table 8.1.

Feature 'identification' represents a change at product-level (Product 4), which requires to propagate this change to the SPL-level, by adding 'identification' from the FL of Product 4 to the current FM, using the feature model refactoring scenario. Red asterisk is placed next to the refactoring points of Fig. 8.5, to indicate alterations in the current FM, which represent the bootstrapped SPL under construction.

- Fourthly, once the bootstrapping activity is completed, we have performed the evolution activity, which encompasses the bootstrapped SPL with the new product (i.e., Product-New). Using the feature model refactoring scenario, we have extended the bootstrapped

SPL with features of ProductNew. As a necessary and preliminary step, we have derived and stored features of the ProductNew in the FL.

The current FM at refactoring point 3 and refactoring point 4 The 'interface_support' new feature represents a change in the product-level. As shown in Fig. 8.6, this change was propagated to the SPL-level (adding new features), which requires to add 'interface_support' feature from the FL of ProductNew to the current FM, using the feature model refactoring scenario. We have used Refactoring 12 'add Optional node' (seeTable 6.2) to refine the current FM at refactoring point 2 (Fig. 8.5), with features of ProductNew. The same scenario was applied for 'flag_3' feature. Fig. 8.6 presents the current FM at refactoring point 3 and refactoring point 4 that models the CSVF resulting SPL. Red asterisk are placed next to the refactoring points of Fig. 8.6, to indicate alterations in the current FM.

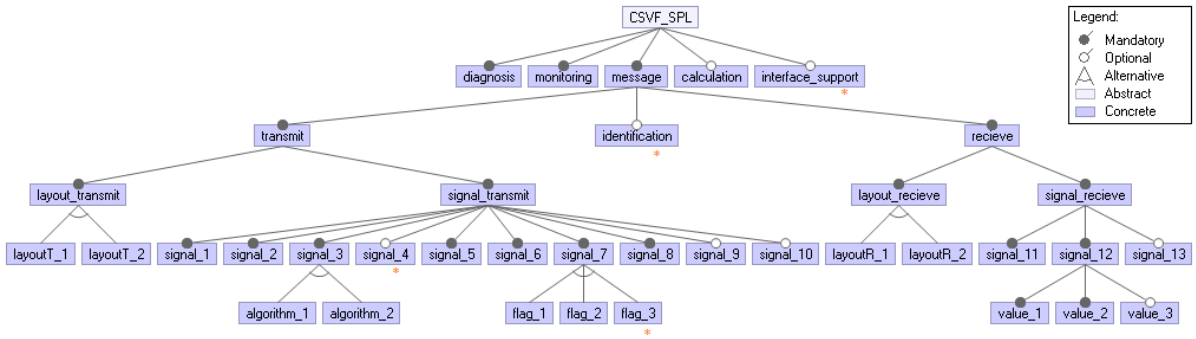


Figure 8.6: The current FM at refactoring point 3 and refactoring point 4 from Table 8.1.

- Fifthly, we have offered a solution to CSDT concerning the most important challenge that the team faces in their daily work (feature location and feature mapping). Thus, we have performed the feature mapping activity. Practically, we have traced the artifacts of CSVF resulting SPL. The solution maps feature of the current FM modelled CSVF resulting SPL to feature-related-code fragments of the code. Then we have used friendlyMapper tool to create the features list, to import the feature-related-code fragments (i.e., routines), to create the traceability links from each feature to corresponding routines, and finally, to maintain the traceability links whenever a feature change occurs. It is worth remembering that the feature mapping activity and friendlyMapper are fully explained with a detailed manner in subsection 7.3.3.

Finally, we have successfully finished the case study design step, by preparing a typical documentation and readable artifacts of the CSVF resulting SPL, as depicted in Fig. 8.3 - Fig. 8.6 and Fig. 7.49. The EvoSPL approach activities have been applied to the CSVF and the current FM has been derived, synthesised, and evolved, as shown in Fig. 8.6. At this point, step 2 of the case study can be started to assess the EvoSPL approach, by the project manager and CSDT.

8.3.3 Execution of the data collection on the case study

To collect the data, we have conducted an empirical study, and we have used specific data collection methods, such surveys, interviews and observations. During the empirical study, we have created a survey. We first have outlined the study; we have defined a set of dependent variables in the field of empirical research criteria and we have declared the hypotheses.

An empirical study When conducting the empirical case study, we have defined a set of items to be evaluated, by asking CSDT to perform specific tasks and then answer questions while they are applying directly the Evo-SPL approach and its related artifacts. Ideally, to compare the developer's solutions with a possibly correct solution, in order to investigate our hypotheses and answer the evaluation questions, we have defined correct solutions for the tasks. For the aim of evaluating our approach, we have defined five dependent variables and formulated the following hypotheses (He – Hypothesis of the empirical study) to measure the 'effectiveness', 'efficiency', and 'correctness' of the EvoSPL approach (the hypotheses are presented in chapter 1).

- He1: the EvoSPL approach is effective.
- He2: the EvoSPL approach is efficient.
- He3: the EvoSPL approach is correct.

Regarding 'effectiveness', we have defined (1) 'effectiveness-SMD', which is calculated as the ratio between the number of correct variability retrieval scenarios from the SMD that the member of CSDT has identified and the total number of the correct retrievals. We have also defined (2) 'effectiveness-FM', which is calculated as the ratio between the number of correct feature retrieval scenarios from the current FM that the member of CSDT has identified, and the total number of the correct retrievals. Furthermore, we have defined (3) 'effectiveness-Evo' is calculated as the ratio between the number of correct evolution scenarios to the current FM that the member of CSDT has performed and the total number of the correct evolutions. Finally, we have defined (4) 'effectiveness-TT', which is calculated as the ratio between the number of correct traceability links retrieval from the traceability tree that the member of CSDT has identified whenever feature changes occur and updating the code accordingly scenarios and the total number of the correct scenarios.

Regarding 'efficiency', we have defined (1) 'efficiency-SMD' as the ratio between the number of the correct variability retrieval scenarios from the SMD that the CSDT member identified and the total time he/she has spent. Also, (2) 'efficiency-FM' is computed as the ratio between the number of the correct feature retrieval scenarios from the current FM that the member of

CSDT has identified and the total time he/she has spent. Furthermore, (3) 'efficiency- Evo' is calculated as the ratio between the number of the correct evolutions of the current FM that the member of the CSDT has performed and the total time he/she has spent. Finally, we have defined (4) 'efficiency-TT', which is calculated as the ratio between the number of correct traceability links retrieval scenarios from the traceability tree that the member of CSDT has identified, and the total time he/she has spent.

Regarding 'correctness', we have defined 'correctness-EvoSPL' as the ratio between the number of positive feedback from the project manager about the validity of the variability information provided by EvoSPL approach (i.e. artifacts) and the number of questions of the survey that is written and dedicated to get the feedback of the project manager. To perform the data collection step, we have walked through the following mini-steps.

Ms1. We have met the members of CSDT (the project manager and seven developers). We have started the first session, by a training that includes a presentation about the EvoSPL approach, regarding both the artifacts and capabilities of our approach. For example, most of the members of the team have no clear idea about feature modelling concepts.

Ms2. We have established the second session; it took four days with eight sessions; we have established a one-hour session with each individual member of the team. We have introduced the objective of the study, artifacts of the EvoSPL approach to be used to conduct the case study, and the tasks.

Ms3. We have performed the empirical case study. In the first 10 minutes, we have taken the information related developer experience and background in a form (see Appendix B.1), and then we have given them 3 tasks to perform, using the EvoSPL approach. Each task consists of 3 tags. The tasks depend on the resulting artifacts of the EvoSPL approach. Practically, the tasks are related to their normal daily work, while they are using the normal approach, but during the evaluation, we have asked them to use our approach. Summaries of the given tasks are contained in Appendix B.2.

We have repeated the analysis for the same tasks, to estimate the 'efficiency' dependent variable, but this time, we have asked the new members of the team, who are under training (have no long experience on using the normal approach), to perform the tasks, using the normal approach and our approach. The normal approach is an ad-hoc reuse-based development process, which is adopted by the company for a long time, to develop the CSVF and satisfy customer's needs.

Our goal regarding the repetition is to evaluate our approach for the task that may take several hours to perform using the normal approach, like finding the feature location in the code, in

order to update its content upon receiving a change. A member with a long term of experience may remember the feature location, because of the long term of using such code. A trainer (new member of the team) is a good candidate to participate in the experiment to evaluate the 'efficiency' of the EvoSPL approach compared to the normal approach.

Interviews Data collection through interviews is important in our case study. Thus, we have performed interview-based data collection (i.e., semi-structured interviews), by asking a series of questions to the members of the CSVF related to EvoSPL approach. We have conducted one interview with every individual member of the team. The interview dialog was guided by a set of questions, which are planned in advance, but they were not asked in the same order as they were listed. We have formulated the interview questions based on the evaluation questions, but they are of course not formulated in the same way. We have prepared and asked open questions (i.e., allowing and accepting a broad range of answers and opinions from the members of the team) and closed questions (i.e., offering a limited set of alternative answers).

We have divided the interview session into a number of stages. First, we have planned the interview and decided whom to interview. Also, we have decided to meet seven interviewees. Due to the qualitative nature of our case study, we have selected the interviewed members based on differences instead of trying to duplicate similarities (e.g., we try to involve members from different roles and personalities).

Furthermore, before the interviews, we have spent much time on studying the available documentations, in order to get a good background understanding. The interviews were open and consisted mostly of encouragement to the interviewed people, to help them explain how things happened using an ad-hoc reuse approach (branching) and our approach.

Second, we have presented the objective of the interview and the case study, and we have explained how the data from the interview will be used. Then, we have asked a set of introductory questions about the background of the interviewed members. After the first stage comes the second stage (main interview questions), which take up the largest part of the interview. We have asked each interviewee and we have written down important information. We have asked the interviewee for an explanation of the points that are related to our approach and baseline approach. Before closing the interview, we have summarized the major findings of the interview, in order to get feedback and avoid misunderstanding.

Furthermore, we have established an interview with the project manager, and we have asked her open questions that are related directly to the correctness and performance of our approach. We have urged her if the EvoSPL approach can cover the demands of their daily normal work, and

if our approach can substitute the ad-hoc reuse approach that has been adopted by the team for a long time.

Observations We have conducted the observations, in order to investigate how the tasks related to the EvoSPL approach are conducted by the individual members of CSDT. Thus, we have joined the company as researchers and for the daily normal working hours. We have monitored a group of the CSVF while they are interacting with our approach, and we have taken notes. Based on the observations, we have analysed the monitoring sessions. The advantage of the observation is that they provide a deep understanding of the evaluation that is studied; we have performed observations in an industrial setting and obtained a high degree of realism.

Survey The main aim of the survey that performed in this empirical study is to generalize the findings of the proposed approach. Thus, we have performed interviews with the members of CSDT who cover whatever target population that we have (i.e., difference skills, experiences, rolls, and personalities). The survey was started when the EvoSPL already has been in use for a while. The primary means of gathering qualitative or quantitative data are interviews or questionnaires. Thus, we have collected standard information from members of the team, by means of a questionnaire. During interviews with each member of CSDT, we have asked them to answer a questionnaire about EvoSPL approach. Then we have analysed results from the survey, to derive descriptive and explanatory conclusions towards the hypotheses and answers for the evaluation questions (section 8.3.4).

We have interviewed the project manager and seven members of CSDT, and we have asked them about (1) what they think regarding the process and resulting artifacts of the EvoSPL approach, (2) how process of the EvoSPL approach can improve the team performance, and (3) if they prefer the EvoSPL approach on the normal approach and why. A sample from all the members of the team is selected at the company. A questionnaire is constructed to obtain information that we needed for the proposed research. The questionnaires are answered by the sample of members of the team. The information collected is then arranged into a form that can be handled in a quantitative or qualitative manner. The questionnaire provided in Appendix C.

Questionnaires During the interview, we have provided both in paper and electronic form (i.e., e-mail) questionnaire. The basic method for the data collection through questionnaires is that we have given the questionnaire together with instructions on how to fill it in, and we have sent it via email. We have asked the responding members to answer the questionnaire and then return it to us. We have compared the on-paper form answers to the electronic version answers that the member has provided, during the empirical study, to make sure the interviewees have a consistent opinion regarding our approach.

We have provided multiple-choice questions towards receiving feedback about our approach and compression to the baseline approach. We have chosen to let interviewees handle the questionnaires face-to face offers a number of advantages: (1) interview surveys typically achieve higher response rates, (2) an interviewee generally decreases the number of uncertain answers, like 'do not know and 'no answer', because he/she can answer questions about the questionnaire, and (3) it is possible for the interviewee to observe and ask questions. The disadvantage is the cost and time, which depend on the size of the sample. Fortunately, in our investigation study, the sample is small.

The survey questionnaire is designed to collect the feedback of the members of CSVF from two perspectives. First, we have asked the member a series of questions that help us to investigate to 'what degree the individual member of the team has a positive feedback about the EvoSPL approach' concerning feature modelling, feature mapping, and transformation to systematic reuse (i.e., SPLs). Second, we have asked the member a series of questions to 'compare the EvoSPL approach to baseline approach' concerning efforts to switch to systematic reuse, variability extraction, evolution, and maintenance, as well as degree of reuse and understandability.

8.3.4 Analysis of the collected data and reporting

The case study conducted in this thesis aims to address the applicability of SPLs adoption in practice and to prove the fitness of the EvoSPL approach in the automotive domain. Thus, an empirical study of usage, benefit and challenge of the EvoSPL approach was conducted, in a real setting, using an industrial-sized products family. The evaluation of our approach focuses on collecting the evidence from the perspective of participants, who are working in the automotive domain. Thus, our study has a qualitative nature. The analysis of the study helps to obtain preliminary feedback from the industrial participants regarding the proposed approach.

We have performed a qualitative analysis related to the dependent variables (section 8.3.3) to prove the hypotheses. The qualitative analysis was undertaken based on the collected data that was performed earlier (section 8.3.3). Table 8.3 summarises the results of the qualitative analysis related to the tasks that were performed by the members of the CSVF within the empirical study.

Firstly, we have investigated the 'effectiveness' of task 1 (effectiveness-SMD), which is related to the task of retrieving the variability information from the SMD. We compared the developer's solutions with the correct solutions. The result reveals that the developers were able to achieve a high percentage; they have solved around 93% of the total tags related to this task. In what concerns the 'effectiveness' of task 2 (effectiveness-FM), which related to using the current FM to retrieve variability information, the CSDT members were able to retrieve correctly around

Table 8.3: Mean and max for the data analysis of the dependent variables.

subjective dependent variable	mean	max
effectiveness-SMD	0.93	1.00
effectiveness-FM	0.89	
effectiveness-Evo	0.85	
effectiveness-TT	0.95	
efficiency-SMD	0.26	0.30
efficiency-FM	0.21	0.25
efficiency-Evo	0.18	0.20
efficiency-TT	0.22	0.25
correctness-EvoSPL	0.86	1.00

89% of the total variability information. The 'effectiveness' of task 3 (effectiveness-Evo), which is related to performing the evolution scenarios of the current FM, revealed that the CSDT members were able to perform correctly around 85% of the total evolutions. The percentage is high also for task 4, which is related to retrieving traceability links from the traceability tree and updating the code accordingly scenarios that the CSDT members were able to perform correctly, it reaches around 95%.

Regarding the tasks that we have asked the new members of CSDT to perform within the empirical study, using both the normal approach and our approach in two separate sessions. The results show that CSDT members have taken, on average around (1) 20 minutes to complete task 1, with 'efficiency-SMD' value of 0.14, (2) 30 minutes to complete task 2, with an 'efficiency-FM' value of 0.08, and finally around (3) 28 minutes to complete task 3, with an 'efficiency-Evo' value of 0.09.

Fig. 8.7 shows the chart that compares the EvoSPL approach against the company baseline (i.e., the normal approach) regarding 'efficiency', while CSDT is performing the tasks related to the empirical study. In total, the comparison reveals that CSDT have performed the tasks related to the empirical case study using our approach more efficiently than using the normal approach.

With regard to 'correctness', we have depended on feedback of the project manager. The project manager highly agreed that the results of the EvoSPL approach (the process and generated artifacts) are valid (i.e., correctness-EvoSPL is equal to 86%). She has confirmed that the current FM is readable and understandable, as well as models the features of the CSVF resulting SPL. She has also affirmed that the traceability matrix is an important artifact that offers a solution to map each feature to feature-related-code fragments of the code consistently, which is a target and urgent demand of the team for a long time.

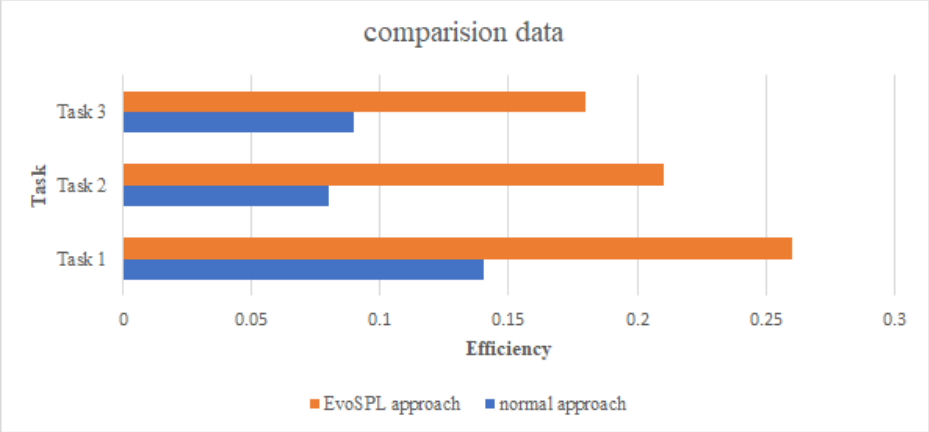


Figure 8.7: The comparison of the efficiency between the EvoSPL approach and normal approach.

The project manager has approved that the current FM can be used as a compact representation of features that belong to the CSVF resulting SPL. Furthermore, it could be widely used during the whole CSVF future SPL development process, as input to produce other products. The project manager has endorsed the current FM as a common artifact to perform the elicitation process that covers needs, requirements, and constraints from different sources, such as customers, users, domain experts, etc. Thus, she has recommended that the current FM can play an important role in requirements engineering for an SPL.

Furthermore, we have compared the current FM that has been defined by the project manager and have drawn by ourselves (depicted in the upper side of Fig. 8.8), and the current FM that has been derived by our approach (depicted in the lower side of Fig. 8.8). The result reveals that our approach has obtained all the features that have been defined by the project manager, the only difference is that the current FM derived by our approach improves and gives more accurate relationships between the features, as confirmed by the project manager. As can be seen in Fig. 8.8, the current FM defined by the project manager is different from the model created by the Evo SPL approach. The places of differences and improvements are numbered from 1 to 5 in lower side of Fig. 8.8 and titled as the relationship 1 to 5 in Table 8.4.

Table 8.4 presents the relationships [1- 5] appeared improved, as we have claimed, in the current FM derived by our approach. The table marked the places that reflect an opinion of the project manager regarding which model shows better definition for the relationships. For example, the project manager has approved that the current FM derived by the EvoSPL approach (depicted in the lower side of Fig. 8.8) improves 80% of the relationships among features, such as, 'relationship 1', 'relationship 3', 'relationship 4', and 'relationship 5'. At the same time, she has seen that

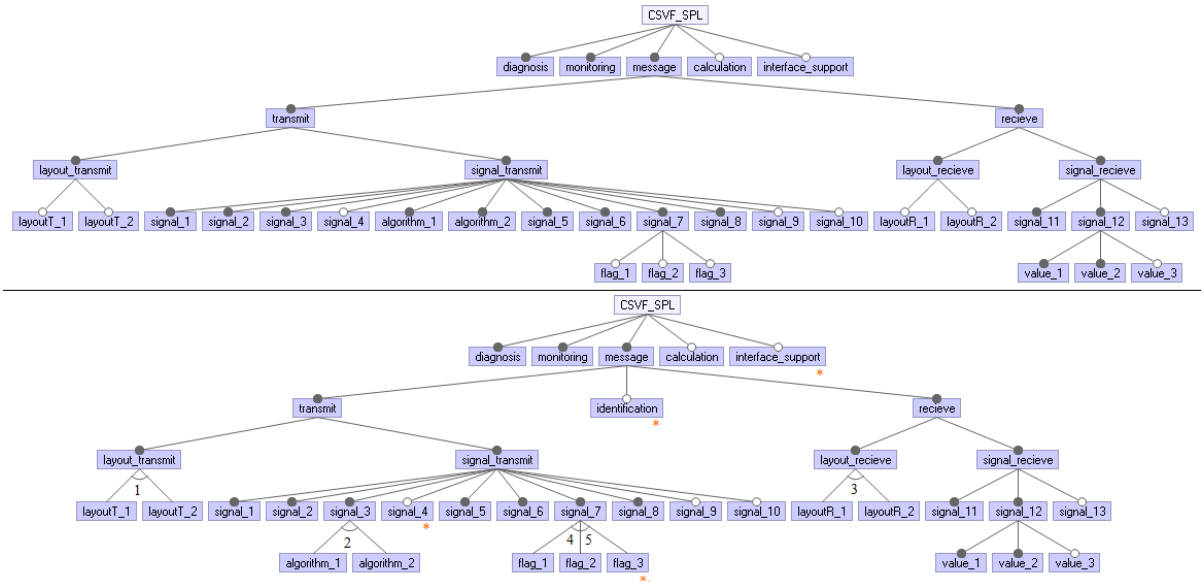


Figure 8.8: The current FMs defined by the project manager and derived by our approach.

Table 8.4: The relationships appeared improved by the current FM defined by the project manager or by the current FM derived by the EvoSPL approach.

The current FM	defined by project manger	derived by the EvoSPL
relationship 1		☑
relationship 2	☑	
relationship 3		☑
relationship 4		☑
relationship 5		☑

the current FM defined by the project manager (depicted in the upper side of Fig. 8.8) presents more accurate definition for 'relationship 2'.

For exploratory purposes, we have returned back to the CAN Matrix and code of the CSVF, and we have investigated the requirement specifications that document 'signal_3', 'algorithm_1', and 'algorithm_2'. We have found that 'signal_3' appears in all the variations of the CAN Matrix, while each CAN Matrix contains the specification of 'algorithm_1' or 'algorithm_2' exclusively. Furthermore, we have investigated feature-related-code fragments of each feature, and we have seen that 'signal_3' has implementation code in all the products of the CSVF, while 'algorithm_1' and 'algorithm_2' has implementation code in all the products also, but one of them is enabled in an individual product, using the variability mechanism adopted by the team. We conclude that the current FM derived by our approach supports an accurate relationship definition among those features ('signal_3', 'algorithm_1', and 'algorithm_2').

We have noticed that during the manual observation of CSDT, while the members were performing the tasks related to the EvoSPL, they were able to understand and interact with the approach smoothly. They have returned a positive feedback; they were praised the activities and artifacts of the EvoSPL approach. They highly agreed that the approach can be used to achieve the intended objectives. They prefer the approach (and not just the feature mapping steps) is supported with semi-automation tool. Fig. 8.9 shows the result of the observation regarding retrieving the feature location of 'signal_13' by the members of the team, upon receiving a change request, to activate this feature. The figure compares the number of the faults in retrieving the feature location of 'signal_13', when the members of the team use the normal approach and EvoSPL approach.

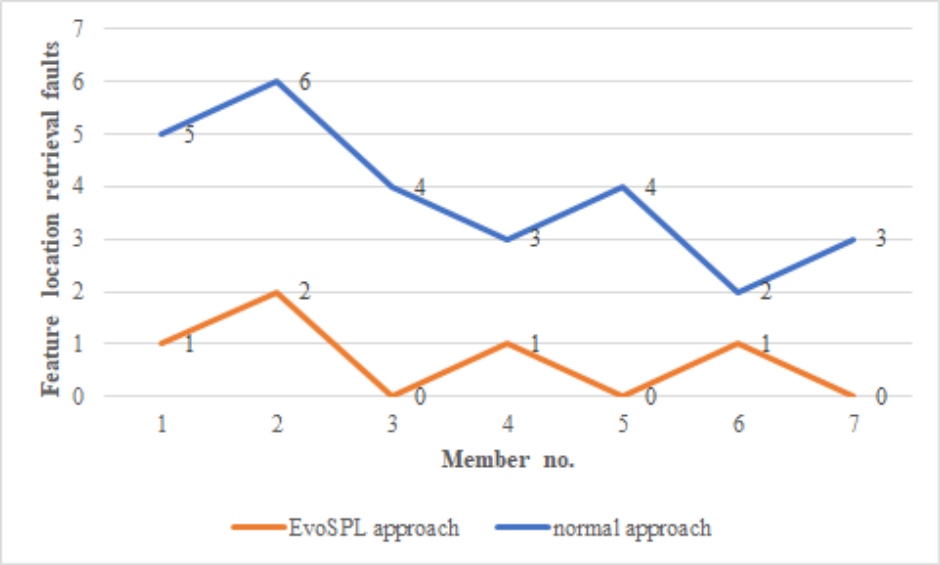


Figure 8.9: The result of the observation regarding faults in retrieving feature location of 'signal_13' by the members of the team.

The chart appeared in Fig. 8.9 displays in the x-axis the member number of the team (i.e., 1 to 7) and in the y-axis the number of the fault attempts that have occurred by the team member while trying to retrieve the location of 'signal_13' in the code. We have chosen to observe the team while they are trying to retrieve the location of this feature, since it took us a lot of time and effort to search for the feature-related-code fragment of 'signal_13' feature. Furthermore, during the interview, we have asked the team members about the location of this feature in the code, surprisingly, we have received different and wrong answers.

From Fig. 8.9, we can observe that the fault retrieval values of feature location are between [0 - 2] of all the members of the team that have used the EvoSPL approach. The values are an indicator of the efficiency of the traceability tree of our approach, which offers a solution to

retrieve feature location accurately, especially with the tool support (friendlyMapper). Also, we can observe that the members of the family have failed to find the feature location of 'signal_3' four times on average. During each single session of the case study interview, the members of CSDT have explained the cons and pros of the baseline approach. Most of the members have agreed (i.e., 'agree' option) to recommend the EvoSPL approach to be used for systematic reuse, while the others have strongly agreed (i.e., 'strongly agree' option). They have expressed their appreciation for the way that the EvoSPL approach offers a solution to map each feature to feature-related-code fragments of the code. They have complained that it has taken them, sometimes , many hours to search for all the feature-related-code fragments in the code, when they are using the normal approach. Almost all the members of the CSVF wish that the process of the EvoSPL approach is semi-automated.

Besides, we have engaged with the project manager with a long interview session. She has clarified that the feature modelling concept is still strange for the practitioners in the automotive domain. She has also explained that the EvoSPL approach (the baseline approach also) can cover the market demands even though of the challenges that the team faced during the use. She has embraced the capabilities of our approach, as well as she has confirmed that the feature mapping is both challenging and practical; she declared that the team needs strongly to adopt the feature mapping process, including the friendlyMapper tool, during the daily normal work.

A training course has been developed in the company, which was unrelated completely to our research work, to help the team to enhance their knowledge and skills at a level that will readily transfer the team to SPLE adoption in the workplace. The project manager has abstracted the findings of the training course and put the book available in our hands.

Concretely, we have analysed the members' feedback, which was collected using a survey questionnaire. The members of the team not only have agreed that our approach could reduce the variability management efforts of the CSVF, but also, they have agreed that our approach could help to reduce the effort compared to the baseline approach.

Fig. 8.10 displays a pie chart designed to show how the CSD, who joined the survey to present their opinion regarding the EvoSPL approach, is divided into various segments (i.e., levels of experience). Each segment of the pie is a particular category within the level of experience. In this way, the chart shows the percentage distribution of the members of CSDT according to their level of experience, in the work. As reported by the chart '38%' of the members are with a 'little experience' and the members with a 'medium experience' are with the same percentage. Regarding the 'significant experience' and 'professional experience', they have the same percentage of '12%'. The project manager is included in this analysis, and she is the only one who has the 'professional experience'.

Figure 8.10: The percentage distribution of the members of CSDT according to their level of experience in the work.

The bar chart of Fig. 8.11 presents the result of a survey for the members of CSDT (presented in Fig. 8.10), the members were asked about the EvoSPL approach. The questions referred to the feedback of CSDT about the EvoSPL approach and comparison of our approach to the normal approach. It illustrates the average of the positive feedback for the questions [Q1-Q10] of each member of the team, regarding our approach and comparison to the normal approach. In this bar chart, the height of the bar indicates the average of the positive feedback for the questions of each member.

It can be seen from the figure that the average of the positive feedback of the members [m1-m7] regarding the EvoSPL approach and comparison of the EvoSPL approach to baseline approach fell slightly from 0.80 to 0.98. However, the average of the positive feedback of the members regarding both criteria is convergent. Overall, by observing the chart, we can see that the members were highly trending to give a positive feedback (i.e., agree and highly agree) in both cases, where the average seems to have levelled off. When it comes to presenting the percentage of the members of the team that were motivated to join the survey, the results of the survey reveal that 57% and 43% of the team members were highly motivated and motivated to join the survey respectively.

For the exploratory objective, we have analysed how the experience of the members affects their feedback regarding the EvoSPL approach. Fig. 8.12 is used to show the relationships between the experience of each employee (x-axis) and average of the positive feedback regarding the EvoSPL approach (y-axis) . The height of the bar indicates the average of the positive feedback of the members of CSDT regarding the EvoSPL approach.

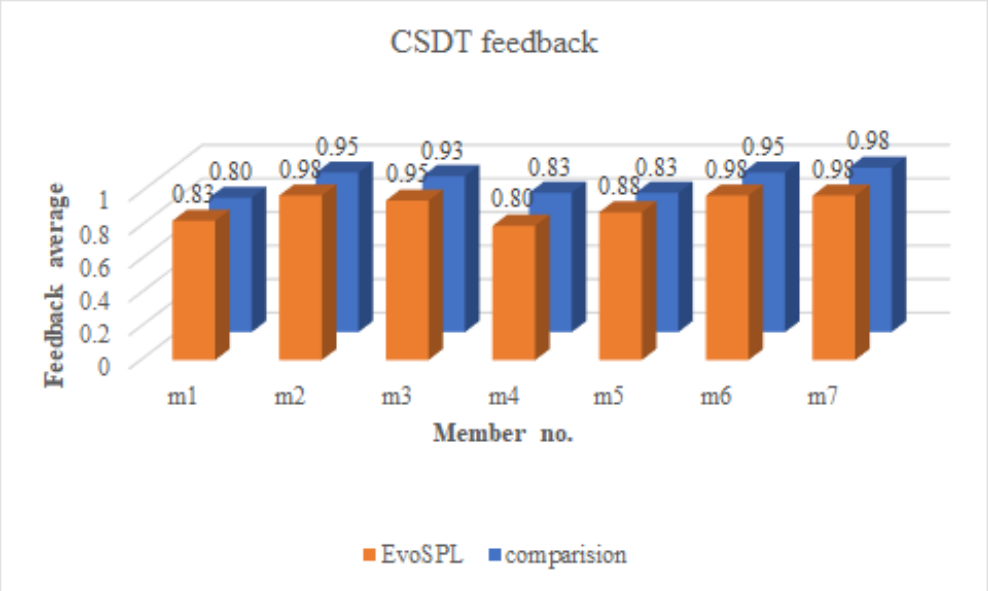


Figure 8.11: The average of the positive feedback of each member of CSDT regarding our approach and comparison to baseline approach.

In general, the graph of Fig. 8.12 reflects that the majority of the team members, with different levels of experience, highly agree to adopt the proposed approach (i.e., EvoSPL approach). Nearly 0.88 (7 out of 8) of the team including the project manager highly recommend to adopt the EvoSPL approach, by a percentage greater than or equal to 92%. Those members are with a 'medium, significant, and professional level of experience'. Nearly, the members with a 'little level of experience' prefer to use the proposed approach with a low percentage difference between the members with other levels, they reach near to 86%.

To analyse the feedback convergence of CSDT for the EvoSPL approach (presented in Fig. 8.11), we have used the standard deviation (SD). The graph of Fig. 8.13 shows the SD of the questionnaire answers performed by CSDT. From the scatter chart, it is clear that the members of the team have delivered converge feedback regarding both the EvoSPL approach (artifacts and process) and comparison to the baseline approach.

In conclusion, the presentation of the data analysis depicted in the charts of Fig. 8.9 to Fig. 8.13 reveal that the members of CSDT were (highly) motivated to conduct the case study, and they have delivered questionnaire answers confirming that they 'agree' or 'strongly agree' to adopt the EvoSPL approach as well as they (highly) agree that the proposed approach reduce the effort compared to the baseline approach. Furthermore, they 'agree' and 'strongly agree' that the EvoSPL is able to perform the tasks and achieve the objectives in an efficient and effective way.

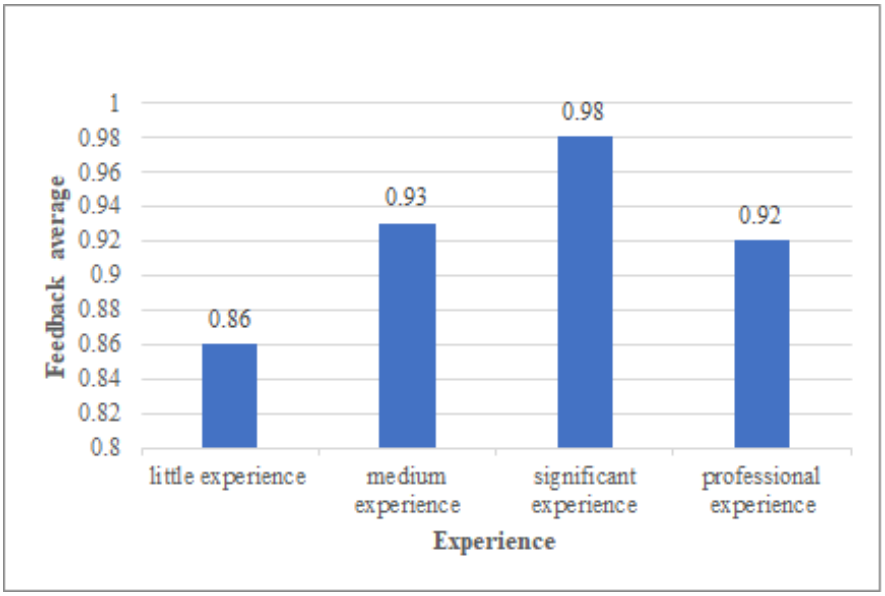


Figure 8.12: The relationships between the experience of each employee (x-axis) versus the average of the positive feedback regarding the EvoSPL approach(y-axis).

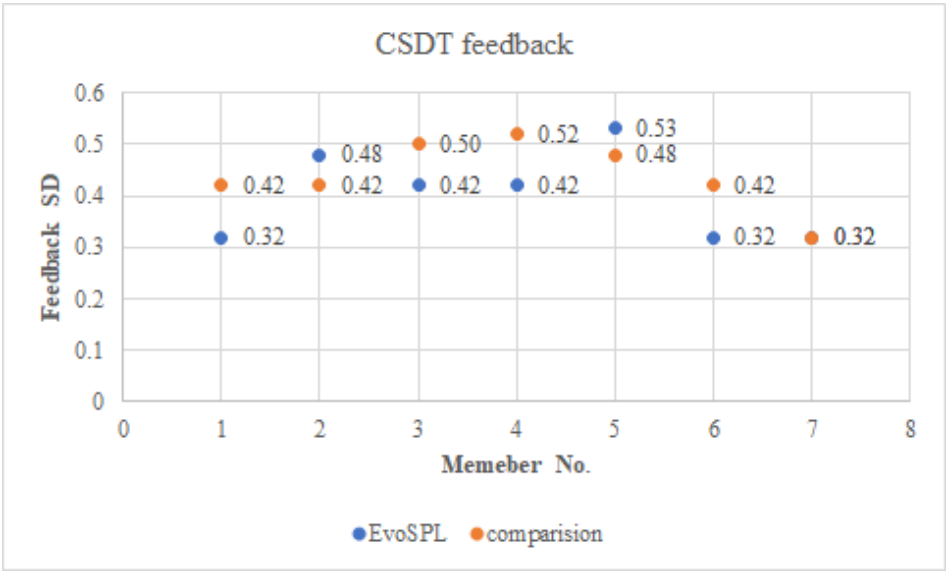


Figure 8.13: The SD of the questionnaire answers performed by CSDT during the survey of the empirical study.

The project manager also has approved the correctness of the EvoSPL approach, including its process and artifacts. Moreover, the survey results reflect a high satisfaction from the project manager and CSDT, which helped us to prove a new hypothesis related to our approach (the

EvoSPL approach is useful). Considerably, all the analysis results reported in this section, including Table 8.3 to Table 7.9 as well as Fig. 8.3 to Fig. 8.13 allow us to accept the hypotheses presented in subsection 8.3.3, which reveal that the EvoSPL approach is effective, efficient, and correct. The acceptance of the hypotheses shows that the EvoSPL approach fits the automotive domain. In the following we present answers of the evaluation questions.

Evaluation question 1. Does the EvoSPL approach derive the current FM that models the CSVF successfully, including, almost, all known features of the family?

According to results of the case study, including the different data collection methods, presented in Table 8.3 and Table 8.4 as well as in Fig. 8.10 to Fig. 8.13, the EvoSPL approach can derive the current FM that models the CSVF successfully, including, almost, all known features of the family.

Evaluation question 2. Is the EvoSPL approach able to refine the current FM with features of (new) products successfully?

According to results of the case study, including the different data collection methods, presented in Table 8.3 and Table 8.4 as well as in Fig. 8.10 to Fig. 8.13, the EvoSPL approach can refine the current FM with features of (new) products successfully.

Evaluation question 3. Is the EvoSPL approach able to map features of the current FM to feature-related-code fragments of the code (both artifacts are related to the CSVF) successfully?

According to results of the case study, including the different data collection methods, presented in Table 8.3 to Table 7.9 as well as in Fig. 8.9 to Fig. 8.13, the EvoSPL approach can map features of the current FM to feature-related-code fragments of the code (both artifacts are related to the CSVF) successfully.

Conclusion As an overall result, the fitness of the EvoSPL approach is successfully evaluated in the automotive domain. The ability to derive the current FM, to bootstrap the products of the CSVF and to evolve a new product to the resulting SPL (using the feature model refactoring scenario), according to the EvoSPL concept and in a comprehensible manner was proven, as well as the possible coverage and automation of feature mapping (see subsection 7.3). However, the feature model refactoring scenario concept supports the creation and handling of an FM refactorings, but it is still possible to face a scenario that is not supported by the feature model refactoring scenario proposed by our approach.

Practically, we have applied the activities (macro steps and steps) of the EvoSPL approach , which are organized in three phases: the reverse engineering phase, forward phase, and mapping phase on the CSVF. Besides we have used the (resulting) artifacts of the EvoSPL approach to conduct the case study and evaluate the findings of our approach. Accordingly, the evaluation questions for the hypotheses were elaborated for the concrete analyses studied in detail. Overally, the results reported in this section allow us to answer the evaluation questions and prove the hypotheses stated in chapter 1, as follows.

- Main hypothesis proposes that it is possible to recommend an approach that supports a re-engineering process to migrate a family of products into an SPL. In addition, it is possible to evolve the SPL after it has been established.
- H 1. proposes that it is possible to develop a semi-automatic difference analysis approach that identifies the differences between the requirements documents of two products that are parts of the same family.
- H 2. proposes that it is possible to specify commonality and variability among the members of a products family in the automotive domain with explicit variability model (e.g., as feature modeling).
- H 3. proposes that it is possible to capture and specify the observed changes in the requirements of existing products of a family in a systematic way, during analysis of evolution of an SPL.
- H4. proposes that it is possible to specify changes in the requirements of a new product during an SPL evolution.
- H 5. proposes that it is possible to apply the appropriate refactoring technique (within an SPL context), to bootstrap an initial SPL from the existing products, and to extend the bootstrapped SPL to encompass another product, on the feature-model level.
- H 6. proposes that it is possible to relate the artifacts that are relevant for variability management, like mapping requirements documents and an FM to its documentation in the reference architecture, and to its implementation in the code, while preventing inconsistencies among them.

Evaluation summary The presented evaluation proves the fitness of the EvoSPL approach, and the evaluation results confirm the overall hypotheses (main hypothesis, [H1-H6], and [He1-He3]). He1 was corroborated as the members of CSDT have performed the tasks effectively by about 90% on average in the industrial case study. Furthermore, He2 was corroborated as the members of the team have shown the capability to perform Task1, Task2, and Task 3 efficiently

at '0.26 out of 0.30', '0.21 out of 0.25', and '0.18 out of 0.20' respectively. Finally, He3 was corroborated by an 86% of correctness according to the project manager feedback.

The hypotheses [H1-H6] were corroborated by the fitness of the proposed concept for the industrial case study, while the members of CSDT are performing the activities of the EvoSPL approach and providing the necessary input. In the industrial case study (interview, survey, and observations), the members of the team from different levels of experience return a positive feedback by about 91% on average, about the fitness of the EvoSPL approach to the automotive domain, especially for the CSVF. Overally, all the hypotheses were proved and the evaluation questions, which were raised to prove the hypotheses, are answered with different data collection methods, during the case study. Regarding the friendlyMapper tool, the evaluation presented in subsection 7.3 allowed us to prove 'correctness' and 'efficiency' of the tool (see Table 7.9).

8.4 Product derivation

As defined earlier in chapter 4, a product is an individual member related to a products family. A product has a collection of artifacts that implement the features of a single software product. Normally, during application engineering, an individual product of a family can be developed, based on the assets provided by the SPL, to address a specific customer needs within the market segment. It is common industrial practice to first derive the initial product from an SPL, then adding and adapting features to satisfy individual customer requirements. These changes are them merged back into the originating SPL [Hin+18].

This section presents the steps for one product derivation from the resulting SPL based on features combination of the product (the features set of a product). Here, a concrete product of the CSVF resulting SPL is defined by a set of features selected and eliminated from the current FM, based on features combination of the product and the feature mapping of each feature to the feature-related-code fragments.

The aim of a product derivation is to collect other evidence to validate our case study. To validate the correctness of our approach, (1) we have derived a product, called ProductDerived, by selecting from the given products of the CSVF resulting SPL, using the current FM, (2) we have compared features combination of the ProductDerived with features combination of each product of the CSVF resulting SPL (see Table 8.1), and finally, (3) we have checked if the features combination of the ProductDerived (features set of the ProductDerived) is a subset of the set of the features of the current FM.

In case of a match between features combination of the ProductDerived and the features combination of one product of CSVF resulting SPL or if the set of features of the ProductDerived is

a subset of the set of the current FM derived by our approach, we can add more evidence to the correctness of our approach. In other words, deriving a valid product from the resulting SPL, using the current FM derived by the EvoSPL approach, validates the findings of our approach. We have supposed that the ProductDerived is requested by a new customer and CSDT have started writing the requirement document (i.e., CAN Matrix) of this product based on the sensor requirements and customer needs. To collect the evidence regarding the product derivation, we go through the flowing steps.

In step1, we have asked the project manager to select from the current FM the required features of the ProductDerived. The ProductDerived supports 'diagnosis', 'monitoring', 'calculation', 'message', 'transmit' with layoutT_1 feature and 'receive' with 'layoutR_1 feature. Additionally, it supports 'signal_1', 'signal_2', 'signal_3', 'signal_5', 'signal_6', 'signal_7', 'signal_8', 'signal_11' and 'signal_12', as well as the optional 'signal_4', 'signal_9', 'signal_10', and 'signal_13', and from the alternative group flag_1, flag_2, and flag_3, it supports flag_1. The ProductDerived uses algorithm_1. Finally, 'signal_12' of the product is a group feature that consists of value_1 and value_2 common features, and one optional 'value_3' feature. We consider always in this section that the 'signals_transmit' and 'signal_receive' features are label features, and they are included by default in features set of each product, if their parents are included. Fig. 8.14 shows the current FM marked by the project manager with the required features (marked with filled circles drawn in the left-upper corner of the rectangles) of the ProductDerived.

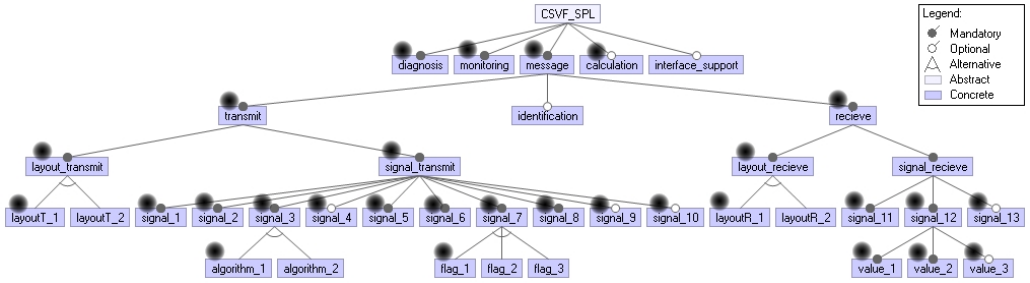


Figure 8.14: The current FM marked with the required features of the ProductDerived.

- In step 2, we have defined the derived product by selecting and eliminating features from the current FM. Fig. 8.15 shows the features combination of one particular product (i.e., ProductDerived) during this step.
- In step 3, we have compared the features combination of the ProductDerived to the features combination of each product of the CSVF resulting SPL (see Table 8.1). Moreover, we have checked if the features combination of the ProductDerived is a subset of the set of features of the current FM.

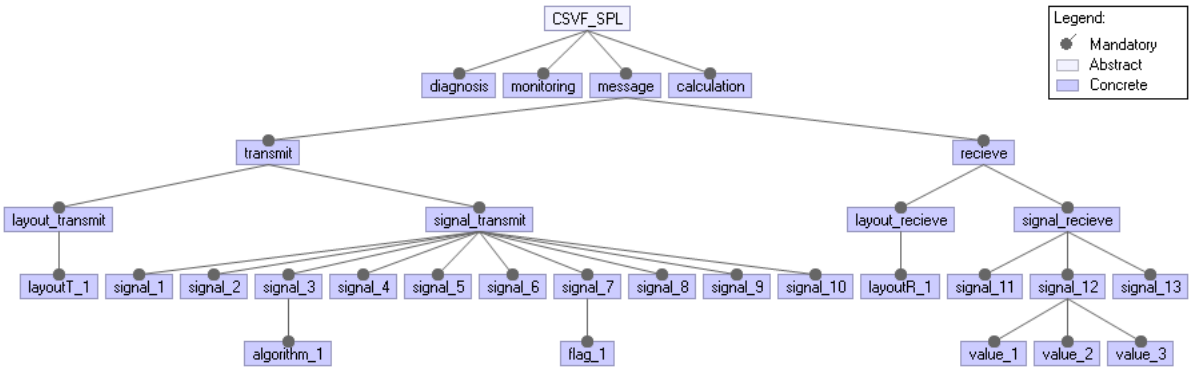


Figure 8.15: The features combination of the ProductDerived.

Features combination of the ProductDerived {'diagnosis', 'monitoring', 'calculation', 'message', 'transmit', layoutT_1, 'receive' , 'layoutR_1, 'signal_1', 'signal_2', 'signal_3', 'signal_5', 'signal_6', 'signal_7', 'signal_8', 'signal_11', 'signal_12', 'signal_4', 'signal_9', 'signal_10', 'signal_13', flag_1, algorithm_1, 'signal_12' , value_1, value_2, 'value_3'} is not equalled to features combination of any individual product of the CSVF resulting SPL, but at the same time, the features combination of the ProductDerived is a subset of the set of features of the current FM. As a result of that, we have successfully derived one particular product of the CSVF resulting SPL that is not supported before by the family, during the steps presented in this section, using the current FM derived and defined by the EvoSPL approach.

- In step 4, we have used friendlyMapper tool, to derive the resulting product implementation (ProductDerived), by mapping each feature in the features combination of the ProductDerived to feature-related- code fragments (i.e., routines) of the implementation code. Fig. 8.16 shows the 'variant derivation' selection from the context menu appearing upon right clicking the 'CSVF' second level item of the traceability tree. The selection of the 'variant derivation' menu item causes the opening of the 'variant derivation' screen (see Fig. 8.17).

Fig. 8.17 depicts the 'variant derivation' screen of the friendlyMapper tool, which enables software engineers to select the features combination of a product (e.g., select features combination of the ProductDerived). The figure shows the selected features (i.e., features combination of the ProductDerived) and eliminated features from the features list of the traceability tree and the 'Get routine' button.

Once software engineers click the 'Get routine' button', the tool automatically presents the

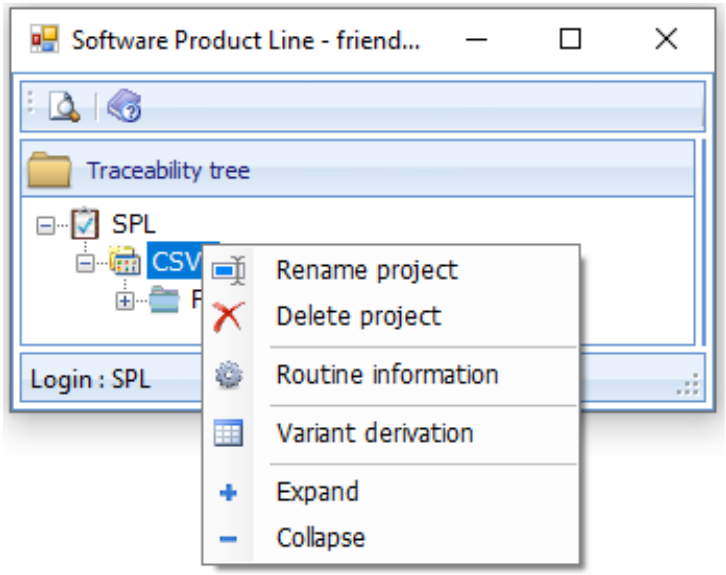


Figure 8.16: The selection of the 'variant derivation' menu item that causes the opening of the 'variant derivation' screen.

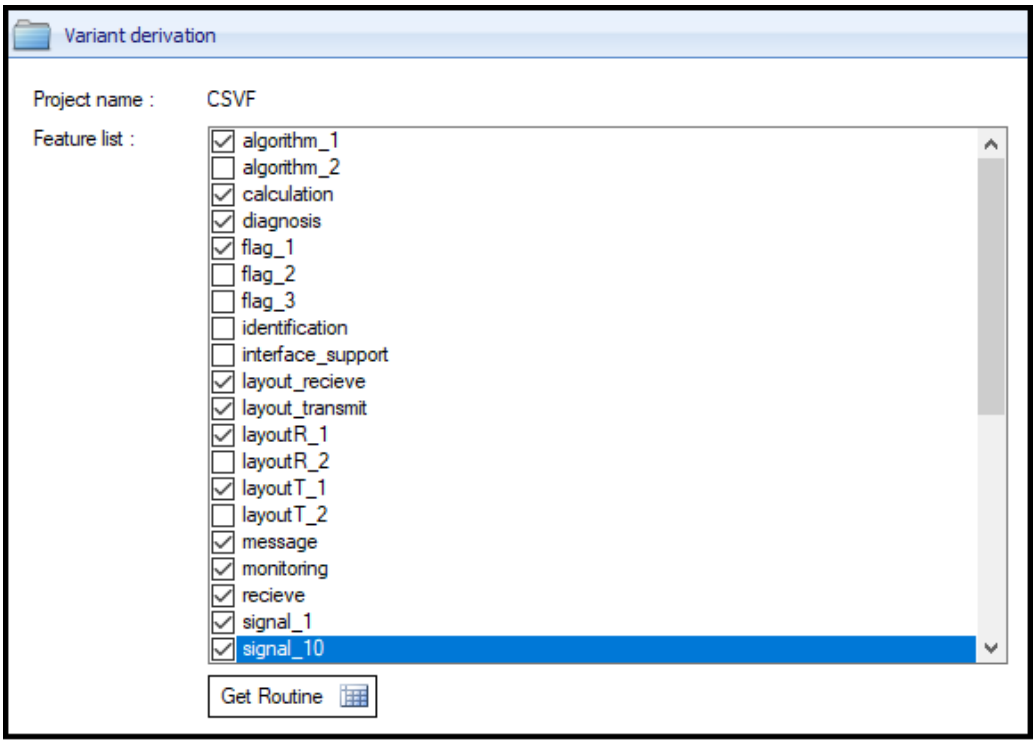


Figure 8.17: The 'variant derivation' screen of the friendlyMapper tool.

resulting product implementation. Fig. 8.18 shows an example for derivation of the Product-Derived created by mapping each feature belonging to the features combination of the Product-Derived to feature-related-code fragments (i.e., routines) of the code, which in total defines the ProductDerived implementation. For readability reasons, Fig. 8.18 presents a partial view of

each feature and its related routines of the ProductDerived.

Figure 8.18: A partial view of each feature belonging to the features combination of the ProductDerived and its related routines belonging to the code of the same product.

8.5 Threat to validity

The validity of a study represents the trust worthiness of the result, to what extent the results are true and not biased by the researchers' subjective point of view. This section describes possible threats to the validity of the evaluation that are relevant to our case study. There are several threats to validity of our approach presented as follows.

1. The industrial case study considers the FM that is with medium size. We believe that the empirical study needs to be conducted on an ideal candidate FM, which includes more features. For example, a product family with more products that have a high variation. However, we believe that the evaluation requires a repository containing product families from different sectors of the automotive domain including, mechanical engineering, production engineering, electrical engineering, electronics engineering and computer engineering.

2. Our approach assigns a name of the feature. This way may be not suitable or should be improved with other techniques to fit all the cases to name the features.

3. The case study performed does not cover all types of the modifications of the current FM supported by the feature model refactoring scenario. For example, in the case study, no modifications for removing a feature from the current FM or converting an Optional feature into Common feature of the current FM. Thus, there is a potential risk of types of feature modifications being not covered by the case study performed but influencing the evaluation results.

4. The interviews and surveys performed had to deal with several challenges representing potential threats to their validity. For example, a higher number of members of the team would have been desirable, but the level of experience limited the number of possible members.

5. During the empirical case study, we have given the members of the team tasks to perform, which may be a source of confusion with new technologies and topics. To cope with this, we have presented a training session prior conducting the case study, to explain the EvoSPL approach and all the related concepts, like feature modelling and feature mapping.

6. The limitation of the study to a single products family (i.e., CSVF) and also to a single development team (i.e., CSDT) present potential threats to its validity . To compensate for this limitation, we have presented a small case study in a form of illustrative example presented at the end of each chapter and we have planned to go on evaluating our approach in other companies.

7. During the data collection, we mostly have asked a single person (the project manager) to evaluate and review the correctness of our approach and noting its artifacts. Besides, we have only invited seven software developers to join the evaluation. In extending this work we should, of course, include more product families and more participants with different levels of working experience.

8. Regarding the estimation of the efficiency of our approach and comparison with the normal approach, members of the team have applied first the tasks with normal approach (first

session), and later they have applied the EvoSPL approach (second session). Repeating the task may affect the time required to execute them. If both tasks are similar, the second task may be done in less time than the required if they make the tasks without previous experience. To reduce this risk, we have performed the two sessions at a different duration of time. We have conducted the first session at the beginning of the month and the second session at the end of the month.

9. During the different data collection methods, we have worried about the fact that the members of CSDT have adopted the same viewpoints regarding the EvoSPL approach and we loss of information. This threat was mitigated by separating the interviews in two sessions and performing each interview separately. The first session was conducted and documented at the beginning of the month, and the second session was conducted and documented at the end of the month.

10. A lack of controlling during the different data collection methods comes with a potential risk of the case study. To avoid this risk, we have planned the case study carefully.

11. Another threat to validity is the fact that the evaluation related to feature mapping requires a manual modification of the code based on the traceability information of the traceability tree of the friendlyMapper tool. May the members of CSDT (i.e., developers with significant experience) use their coding experience to modify the code instead of the traceability information of the traceability tree.

12. Finally, threats to validity compromise our confidence in stating whether the study's result is applicable to other products family, SPLs domains, development teams, and companies.

Chapter 9

Conclusions and future trends

This final chapter concludes this thesis. It closes the thesis by returning to the aspects which were presented in chapter 1. It summarizes the focus of the EvoSPL approach and draws conclusions from the discussions and results in the previous chapters. A summary of the importance and the main contributions of this research are presented in section 9.1 and section 9.2, while Section 9.3 contains suggestions for future research.

9.1 The importance of the thesis

Companies normally adopt designing families of products. Such designing has proved to be effective and economic while satisfying a variety of customer needs [DJT01]. When customers request for additional features, existing products are adapted to fulfill the new requirements. As the number of products increase simultaneous maintenance of a typically large number of individual products is required. In this scenario, as the number of features increases so does the complexity of their development and maintenance, hence a systematic reuse approach is necessary.

An SPL aims to support the development of a whole family of products through systematic reuse. Fortunately, SPLs seek to achieve gains in productivity and time to market. Giving the scenario explained above, dealing with SPLs begin after companies find themselves with a successful products family in a domain. To switch to an SPL approach using an existing set of similar products, variability management is a major challenge, especially in the automotive domain, practitioners lack a re-engineering approach that helps to adopt SPLs.

Moreover, there are three main factors at play in SPLE: 1) commonality and variability management that eliciting and communicating commonality and variability in requirements in form of a variability model; 2) traceability of commonality and variability from the requirements to the code; and 3) managing and tracking reuse of code across the members of a products family, usually driven by the previous two elements. The EvoSPL approach presented in this thesis

introduces an end-to-end re-engineering approach that explores into the factors to accumulate a solution that satisfies all of them (factors). Additionally, it helps to adopt SPLs in the automotive domain. Thus, our approach is designed to deal with requirements documents of this domain.

SPLs evolution exhibits high complexity due to the variability and the interdependencies between products. The EvoSPL presented in this thesis is an evolutionary approach for adopting an SPL [Bos02]. It evolves the SPL based on family of products, by initially migrating the products towards the SPL (i.e., the initial SPL) and then reactively adapting the newly created SPL (the resulting SPL) to encompass other new products. It has the major benefit as the evolutionary approaches of the shortened time-to-market for new products or product variations. It is designed specially to fit small and medium-sized companies, which often rely on existing products as they are not able to make big upfront investments as required by a revolutionary SPL approach (e.g., [MRR04]).

The EvoSPL approach provides software engineers, especially those who desire to adopt a systematic approach to address the evolution of SPLs. This thesis assists (using our approach) the software engineers in the automotive industry in performing the following tasks.

1. Capturing commonality and variability that specify all the products of a family in terms of features (i.e., at the requirement-level).
2. Deriving the FM of the SPL under consideration.
3. Supporting the evolution of an SPL, including the requirements artifact.
4. Addressing the mapping within the context of SPLs, like linking features to their locations in different types of artifacts of an SPL (e.g., implementation code).
5. Adopting a tool that consistently maps between features and the code, in order to automate an SPL mapping.
6. Performing a complete set of evolutionary operations of the FMs (i.e., add, update, and delete).
7. Putting in practice the industrial case study, which handled the evolution and variability of an SPL in the domain of automotive systems, in practice.

SPLs case studies are quite popular, but (1) few industrial-sized case studies are publicly available [bastarrica2019software], and (2) few of the proposed techniques offer a tool support. Thus, the EvoSPL approach presented in this thesis offers a solution to compensate for the shortages

that are mentioned in (1) and (2). Additionally, we have performed an industrial-sized case study at Bosch Car Multimedia company, where the software engineers at the company believe that the EvoSPL approach provides a solution to make it easier to achieve the tasks mentioned above. Also, we have developed the friendlyMapper tool to support the feature mapping process that is supported by our approach.

9.2 The contributions of the thesis

The thesis presented in this work is an industrial research describing an approach where we handle the evolution and variability of SPLs in the domain of automotive software engineering. We presented an evolutionary approach for adopting SPLs, called EvoSPL. Our approach accepts as input the NL requirements documents, reference architecture, and code of a products family. We assumed that the products of a family are generated and managed by using the initial release product and adapting it to satisfy customer needs. EvoSPL approach is based on several techniques, like difference analysis, text-based parsing, feature mapping, and traceability links.

We have implemented our approach and evaluated its artifacts result on an industrial-sized case study at Bosch Car Multimedia company. The results of this evaluation revealed that most of the features of the CSVF are identified and mapped to their places in the code, using the reference architecture as acentric point (see chapter 8). Additionally, the EvoSPL approach models the evolution of CSVF resulting SPL as a sequence of the current FMs. The EvoSPL approach uses FIM and FNM to identify features and give them a proper name. Those methods are proposed and designed specifically for the EvoSPL approach.

Thus, we have concluded that EvoSPL approach offers an end-to-end re-engineering solution for feature identification from the requirements documents and bootstrapping of an SPL from a products family, as well as upgrading the resulting SPL to encompass new products. We also believe that EvoSPL approach guarantees the consistencies among the artifacts of the resulting SPL, by supporting feature mapping.

The main contributions of our approach are presented as follows.

(I) Foundations We have introduced our research work in the context of SPLs, namely SPLE and product families. We have clarified the research objective and motivation of this work. Additionally, we have defined the concepts, challenges, and hypotheses of the proposed approach. Finally, a short outline of the thesis and the structure of its chapters was given (chapter 1: introduction).

We also have presented the background needed to understand the steps and techniques used in our approach: re-engineering, difference analysis, text-based parsing, variability management, SPLs refactoring and traceability. We used these techniques as a basis for our approach. Finally, we have presented techniques and concepts related to our approach and used during the evaluation (chapter 2: background).

In order to find our position between the selected and related work presented in the context of SPLs, we have given an overview of the related works and approaches for the evolution of SPLs. At the start of the related work chapter, we have stated that an SPL evolves when there are changes in (1) the requirements, (2) the family structure or (3) the technology being used. We have reviewed the techniques, approaches, and research works that have been proposed for building an SPL from an existing products family. First, we have introduced and discussed the challenges, types and main approaches of SPL evolution, and we have presented an industrial example for better understanding. Then, we have discussed and given an overview of the main approaches related to migration towards SPLs and the existing approaches for reverse engineering FMs. Moreover, we have introduced refactoring and mapping in the context of SPLs. Evolution of SPLs require mapping features to their implementation, and this problem is addressed by the feature location. Thus, we have explained the related work of this topic (chapter 3: related work).

We have concluded that our approach has to offer a solution to some of the deficiencies in the approaches applied in the SPLs evolution generally and automotive domain specifically: (1) there are no any publicly available industrial-sized case studies, which works a starting point to adopt SPLs in the industry probably one the causes of low adoption of SPL approaches in industry, (2) there are a lack support of textual nature of requirements formats . Thus, there is a clear need for developing an approach that support SPLs evolution (in the automotive domain specifically) using requirements artifacts (using textual nature of requirements formats), and finally (3) there are a lack of end-to-end approach that supports a re-engineering process to extract an SPL from the existing products family and upgrades later the extracted SPL to encompass new products.

(II) The EvoSPL approach To tackle the challenges of evolving an SPL that takes the existing products of a family into account (re-engineering), we have proposed the first contribution of this thesis, which is an evolutionary approach, named EvoSPL. The EvoSPL is a 3-phase approach aims to adopt an SPL in the given context. It considers a process which evolves an SPL from the existing products of a family and focuses on migration of the existing artifacts (namely requirements documents). The basic idea of our approach is to model an SPL and its evolution, by focusing on and descripting the commonality and variability in the requirement documents

of a products family. Thus, in EvoSPL, evolution of an SPL is represented as a sequence of FMs at different points. At each point, the approach refines an FM with the features of a (new) product (chapter 4: EvoSPL approach).

In phase 1 (reverse engineering), we have proposed a step-wise reverse engineering process that exploits commonality and variability across products of a family at requirements-level, to apply the difference analysis and variability analysis techniques. To reduce the complexity of deriving an FM from the products of a family, we contribute to initially use two products of a family, in the derivation process. In this phase, we proposed two novel methods, namely, FIM and FNM, based on several techniques, such as text-based parsing, inference rules, and semantic roles in order to contribute in providing a solution for the feature identification and documentation (giving names) in a collection of products of a family. In our FIM, each feature specification corresponds to a set of ARs. In this phase, we have organized the identified features in the current FM. The model is constructed using a tree, which contains Mandatory features, Optional features and feature groups (and, or, xor groups). In this phase, we have derived the current FM that models the initial SPL successfully (chapter 5: EvoSPL: reverse engineering phase).

In phase 2 (forward engineering), we have presented a step-wise forward engineering process that contributes in bootstrapping the remaining and existing products of a family into the bootstrapped SPL and then extending the bootstrapped SPL with a new product, to deliver the resulting SPL. In this phase, we have proposed the feature model refactoring scenario, which is contributes in refining the current FM with features of (new) products when required. The scenario adopts a set of refactoring notions that are important for safely evolving the existing SPL (the initial, bootstrapped, or resulting SPL), by simply improving its design or even by adding (new) products while preserving the existing ones.

Additionally, the scenario contributes in using several change operations on the current FM, include adding a new feature, eliminating a feature, or changing a common feature to an optional feature and vice versa. In this phase, we have combined the remaining set of the existing products to the initial SPL to deliver the bootstrapped SPL, and then we have evolved the bootstrapped SPL to encompass new products using feature model refactoring scenario successfully (chapter 6: EvoSPL: forward engineering phase).

In the third phase (mapping phase), we have offered a solution for tracing features and code using the reference architecture as an intermediate artifact. The feature mapping activity defines a set of tracing rules to trace features of the current FM to feature-related-code fragments of the code. The basic idea of our solution is to trace each feature to its parts of the most upper layer of the reference architecture and then traces from the identified part of the reference architecture

to corresponding units of the code. In this way, only that units of the identified parts (instead of the entire code base) are analyzed to find the code fragments that are responsible for the feature implementation. This leads to identifying the code fragments that implement the feature and store them in the traceability tree. All this requires defining the ARs of each feature, which involves a backward mapping from features of the current FM to ARs of the FL. This is another contribution of the mapping phase.

We have supported the feature mapping activity with a tool, called friendlyMapper, which could semi-automatically relate each feature belonging to the current FM to code-fragments belonging to the code, as well as update their conformance whenever feature changes occur. The friendlyMapper uses the traceability tree, which contains a set of traceability links, where each link relates a feature of the current model to feature-related-code fragment (chapter 7: EvoSPL: mapping phase).

As an illustrative example, we have considered the ATM products family, which allows the user to perform different kinds of ATM transactions. We have illustrated the main phases of our approach using the family. We have applied the steps of EvoSPL approach on the product of ATM family (chapter 4 to chapter 7), and we have achieved the following contributions.

1. We have presented the ATM products family example. We have developed the products by clone-and-own technique, based on the initial release product (section 4.11: an illustrative example: ATM products family).

2. We have applied the steps of the reverse engineering phase on two products of the ATM family, and we have derived and constructed the current FM, which models the ATM initial SPL (section 5.4: an illustrative example of the reverse engineering phase: ATM products family).

3. We have applied the steps of the forward engineering phase on the remaining product and new product of the ATM family. We have bootstrapped the existing product (Product 2) into the bootstrapped SPL, and we have adapted the bootstrapped SPL to encompass a new product (ProductNew) of the ATM products family, to deliver the ATM resulting SPL (section 6.4: an illustrative example of the forward engineering phase: ATM products family).

4. We have used feature model refactoring scenario to refine the current FM with features of Product 2 and the ProductNew (section 6.4: an illustrative example of the forward engineering phase: ATM products family).

5. We have successfully presented the evolution of the ATM existing SPL as a sequence of the current FM at different points, as follows (section 6.4 an illustrative example of the forward engineering phase: ATM products family).

- Point 1. The current FM that is already derived and constructed, using two products of the ATM products family, namely Product 1 and Product 3.
- Point 2. The current FM that is refined with the features of a product of the ATM products family, namely Product 2.
- Point 3. The current FM that is refined with the features of a new product planned to be added to the ATM products family, namely ProductNew.

EvoSPL contribution The following list summarizes the most important scientific contributions of the EvoSPL approach.

1. The EvoSPL approach provides a novel step-wise process to manage SPLs evolution in automotive software engineering.
2. The EvoSPL approach manages evolution of SPLs assets at requirements-level, specifically, the requirements of the automotive domain, which has a high complexity nature. The approach uses requirements documents written in NL. As previous works have shown great potential in requirements as artifacts for SPLs and to choose them for reverse engineering is viable.
3. The EvoSPL approach identifies features of a products family and synthesizes them in the current FM, which models the initial SPL. The approach contributes two novel methods, namely, FIM and FNM to perform this achievement.
4. The EvoSPL approach contributes to the pending challenge of an SPL evolution through refactoring. The EvoSPL models evolution of the existing SPL using a sequence of the current FMs, at different points. Each point refines the current FM with features of a new product.
5. The EvoSPL contributes the feature models refactoring scenario that safely refine the current FM with features of new (products).
6. The EvoSPL approach contributes bootstrapping steps that involves bootstrapping the existing products into the bootstrapped SPL and evolution steps that involve extending the bootstrapped SPL to encompass new products, to deliver the resulting SPL of the given context.

7. The EvoSPL contributes a solution for tracing features and code (features mapping) after only derived features from requirements.

8. The EvoSPL addresses traceability explicitly along evolution of the SPL, i.e., how the evolution of features relates to architecture and code implementation in SPLs.

9. The EvoSPL approach contributes a feature mapping tool, named friendlyMapper. The tool defines and maintains relationships between the features and code. The tool (1) reads features of the current FM, (2) relates each feature to feature-related-code fragments (i.e., routines), and (3) stores the features and feature-related-code fragments relationships in the traceability tree in forms of traceability links. Each traceability link maps a feature to the feature-related-code fragments. The tool also updates the traceability links whenever feature changes occur.

We have concluded that it is possible to develop an end-to-end re-engineering approach that helps in managing the evolution of SPLs, in a setting where the domain architecture (i.e., reference architecture) is common to a family of products, which was developed without considering a systematic approach. Additionally, we have concluded that it is possible to derive the current FM at requirements-level of a products family, using requirements documents written in NL. Moreover, we have concluded that it is possible to design a feature model refactoring scenario to evolve an SPL, including the current FM, with features of (new) products. Finally, for this part, we have concluded the possibility of offering a solution that maps features to code, using the reference architecture as a centric point. The use of friendlyMapoper helps to preserve consistency between artifacts of the resulting SPL, namely, the current FM and code.

(III) Evaluation In this part, we have conducted an empirical case study to validate our approach. The industrial-sized case study that has been performed in the automotive domain, at Bosch Car Multimedia company, implements an SPL evolution from a set of existing products family (i.e., CSVF) at requirements-level and then maps this evolution to the code-level. The case study has proven the efficiency and the effectiveness of our approach in performing the following.

1. Identifying the specification (using FIM) of a feature as a set of requirements (i.e., ARs), which enable a clear understanding of the features of the CSVF and facilitate the working of developer and testing teams.

2. Documenting the derived feature specifications by assigning a name (using FNM) for each feature and storing them in a specific artifact (i.e., a FL). The features documentation with the related requirements enables a quick understanding of the feature functionality and explains the role of this feature in a products family.

3. Reverse engineering of the current FM based on the identified and documented features of two products of the CSVF. The derived current FM, which models the CSVF initial SPL, assists the CSDT for a better understanding of the members of the CSVF, features, and constraints between features.

4. Bootstrapping the remaining and existing products of the CSVF into the bootstrapped SPL and evolving the bootstrapped SPL with a new product, to deliver the CSFV resulting SPL. Evolving the resulting SPL with new products is supported when it is required.

5. Presenting a solution for mapping the artifacts of the CSVF resulting SPL, by tracing features of the current FM to feature-related-code fragments of the code using the reference architecture as an intermediate artifact.

6. Using friendlyMapper tool to define and maintain the traceability links between a feature of the current FM and feature-related-code fragments of the code (i.e., routines) in the traceability tree.

7. In our thesis, we only have applied a small case study and a single empirical case study. The former is an illustrative example of the ATM product family. The latter is limited to the automotive domain (i.e., CSVF) and to a single development team (i.e., CSDT). This represents a threat to external validity. This aspect of validity limits our approach concerning to what extent it is possible to generalize the findings, and to what extent the findings are of interest to other people outside the conducted case study. There are other threats to validity that are mentioned in the evaluation chapter (Chapter 8).

Finally, We have concluded that it is possible is to propose an approach that helps to adopt SPLs in the automotive industry, as well as, it is possible to help companies in the automotive industry (1) to evolve a given SPL, focusing on migration of the existing products of a family, which were delivered to customers in the past, into the bootstrapped SPL, (2) to use the bootstrapped SPL as a base to evolve it with a new product, to deliver the resulting SPL, and (3) to evolve the resulting SPL with new products when required. Finally, We have concluded that this thesis delivers successfully an evolutionary approach (called EvoSPL) that fits SPLs and provides an industrial-sized case study (conducted in automotive engineering systems), which can support and work as a start point to adopt SPLs in the automotive industrial domain.

9.3 Future trends

This section presents recommendations for the future work concerning re-engineering of a products family into an SPL and supporting its evolution. The software industry has shown a growing

interest in the concept of an SPL. For this, we believe that our proposed approach is interesting to explore and improve.

1. In future work, we aim at applying our approach in additional case studies, in order to assess the benefits and limitations of our approach, in additional product families and different industrial domains. Moreover, from these case studies, we can propose more consistent steps of our approach and more FM refactorings that fit all the possible modifications on the current FM. Besides, we want to improve/enrich the approach by reducing the number of the activities and steps within each activity.

2. Our next obvious step is to build a tool to automate the manual steps of EvoSPL approach. Specifically, we plan to build an integrated development framework for our approach, called EvoSPL framework IDE, which contains a base for satisfying the hypotheses of our approach. The future framework includes difference analysis, variability analysis among textual requirement documents, generation of a FL of each product, construction FMs, and name suggestion during feature identification. Moreover, we plan to build a functionality of the IDE framework that can find and display the dependency-relationships among features of the current FM. In addition, we plan to build an FM refactoring technique around the framework environment, which is able (1) to find the possible refactoring scenarios to refine the current FM with features of a (new) product and after that (2) to perform a transformation for the current FM that improves the quality of the model and maintains its structure to reach the optimal configuration.

   Moreover, we plan to support the framework with a functionality that involves bootstrapping a set of existing products into an SPL and extending the SPL to encompass a new product. Feature location in the context of product families is an important technique to support the systematic reuse. Thus, we plan to support the EvoSPL framework IDE with all the tasks for feature locations. Furthermore, we want to build the framework that can trace each feature of the current FM to the feature-related-arch-elements (i.e., feature-related-subsystem and feature-related-component) of the reference architecture and feature-related-code fragments of the code, and automatically update the feature-related-arch-elements and feature-related-code fragments to maintain their conformance when feature changes occur. We then intend to validate the effectiveness of the future framework on real-life product families from our industrial partners.

3. As defined earlier in this thesis, an SPL is used to represent similar products with multiple variants. It plays an important role in minimizing cost and utilization of the resources. An FM is used to represent SPLs. However, determining the optimal features selection in an NLP, especially for the huge families, is a hard problem [Kum+18] [Guo+11]. As far as it concerns our long-term future research directions, we plan to apply genetic algorithms for

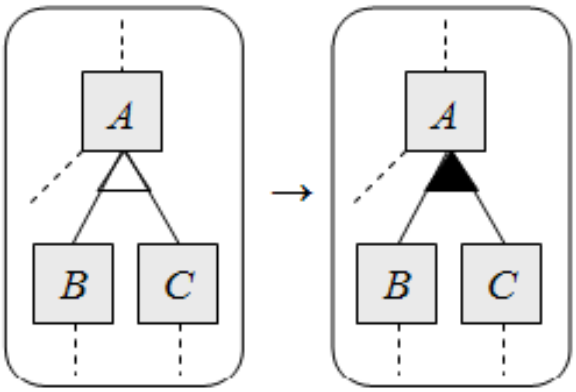
feature optimization in SPLs. Further, we plan to perform a construction of the FM for those features. We plan to evaluate the use of genetic algorithms for feature optimization in SPLs in real-world FMs with different sizes.

Appendix A

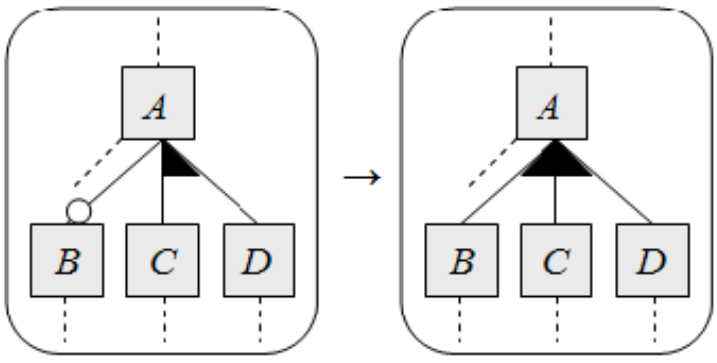
Refactorings catalog

A.1 Unidirectional refactorings catalog

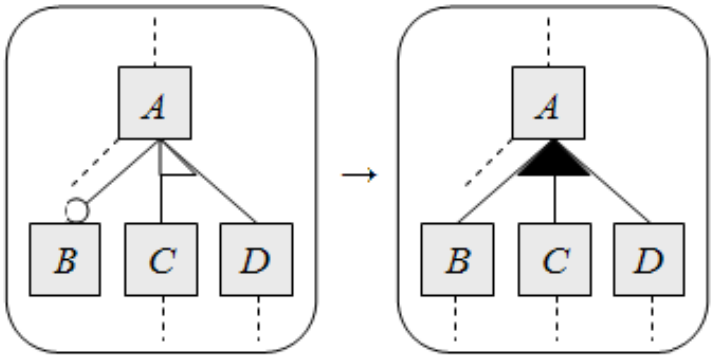
Refactoring 1. *⟨convert alternative to or⟩*



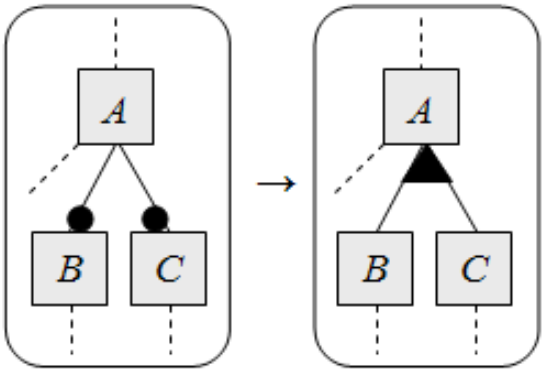
Refactoring 2. *collapse optional and or*



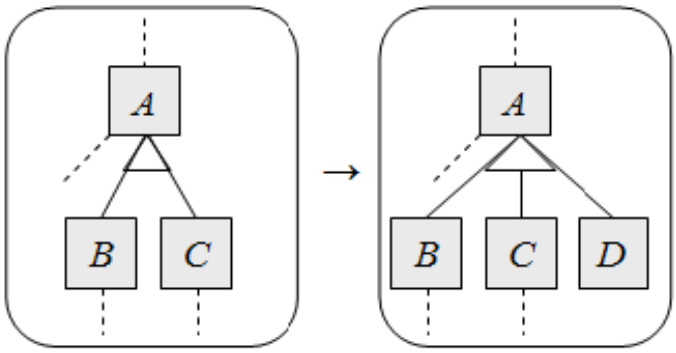
Refactoring 3. *collapse optional and alternative to or*



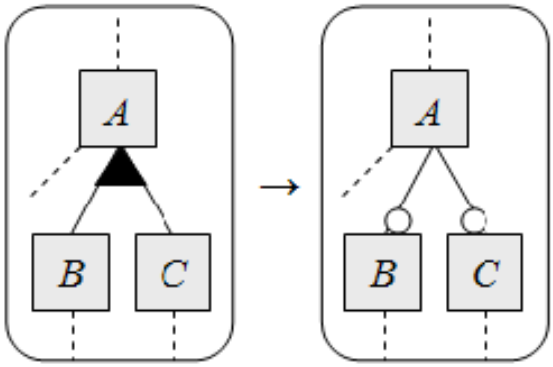
Refactoring 4. *⟨add or between mandatory⟩*



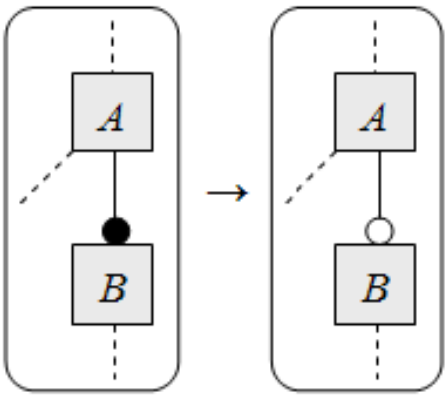
Refactoring 5. *add new alternative*



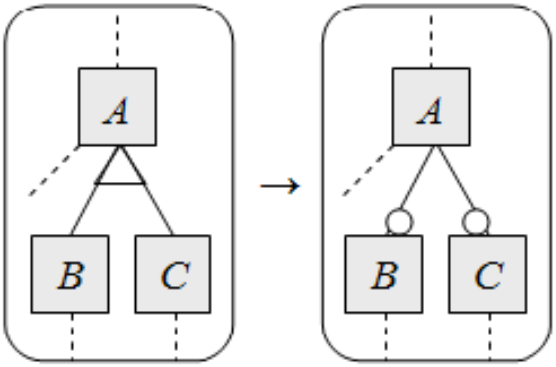
Refactoring 6. ⟨*convert or to optional*⟩



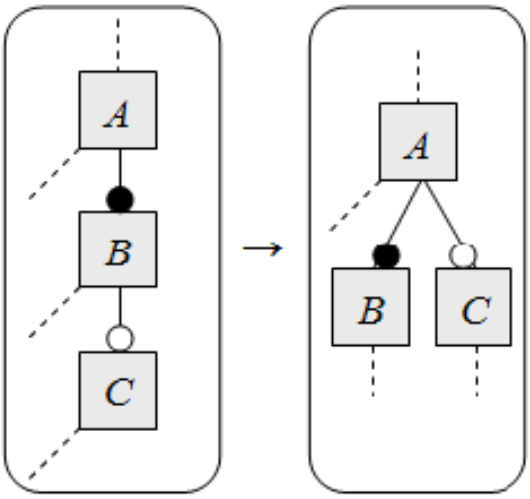
Refactoring 7. ⟨*convert mandatory to optional*⟩



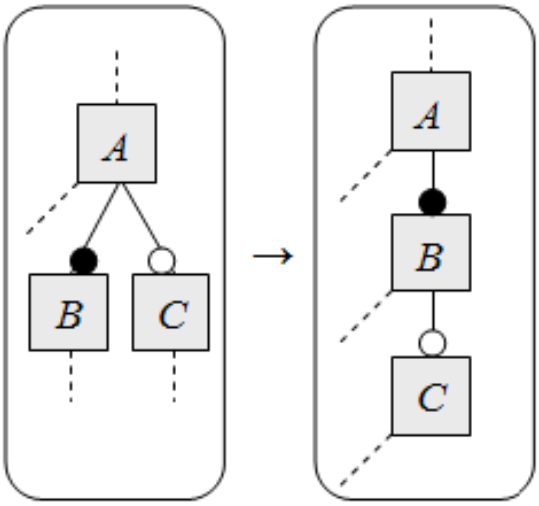
Refactoring 8. ⟨*convert alternative to optional*⟩



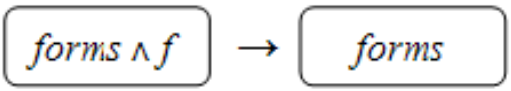
Refactoring 9. ⟨*pull up node*⟩



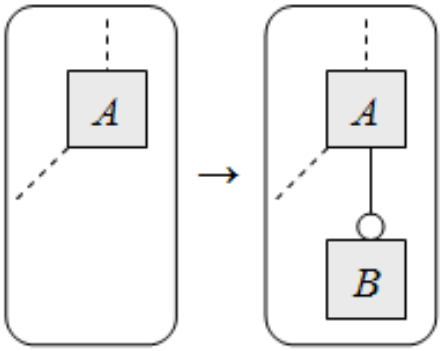
Refactoring 10. ⟨*push down node*⟩



Refactoring 11. ⟨*remove formula*⟩

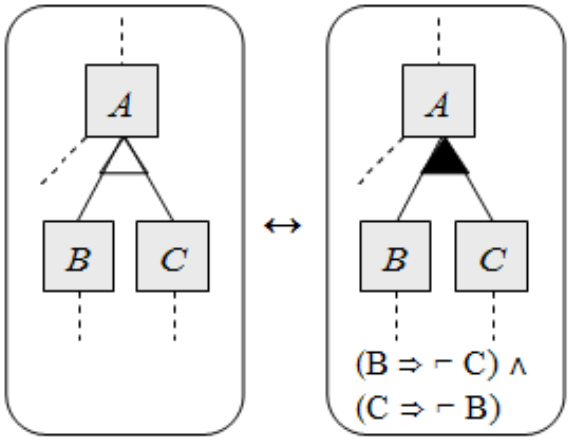


Refactoring 12. *⟨add optional node⟩*

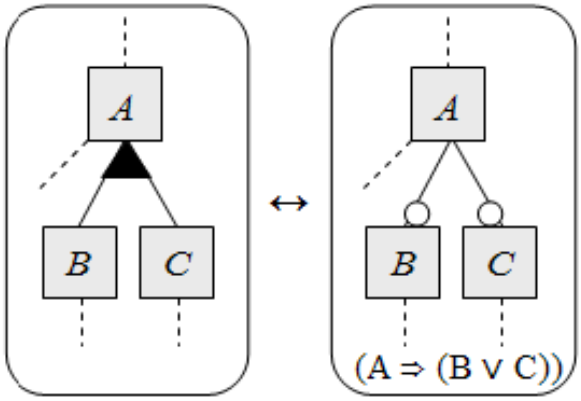


A.2 Bidirectional refactorings catalog

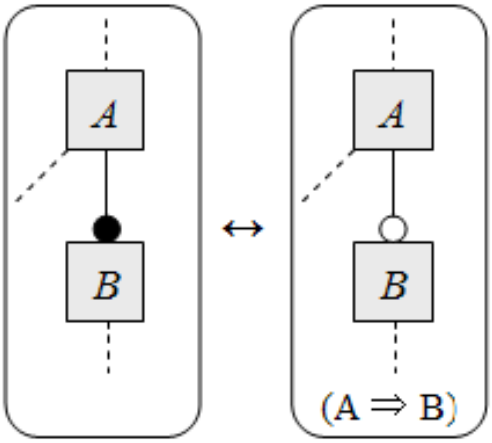
B-Refactoring 1. ⟨*replace alternative*⟩



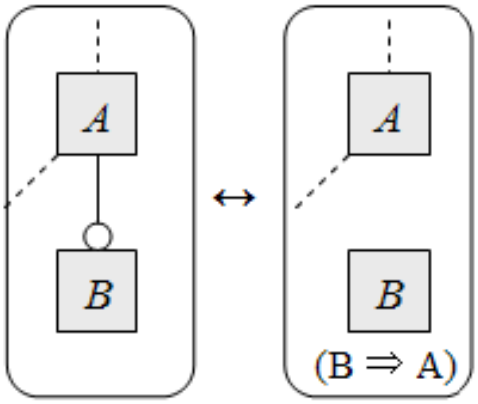
B-Refactoring 2. ⟨*replace or*⟩



B-Refactoring 3. ⟨*replace mandatory*⟩



B-Refactoring 4. ⟨*replace optional*⟩



Appendix B

Industrial Case Study

Survey on evaluation the EvoSPL approach

Karam Ignaim

Classical Sensor Team

- Jana Seidel (Project Manager)
- João Santos
- Hélder Vilas Boas
- Joao Cardoso

21 September, 2018

Braga

Acknowledgment

I would like to thank Bosch Car Multimedia (the Bosch company at Braga) in Portugal and our colleagues t

Special thanks to the Classical Sensor Software Team: João Santos, Hélder Vilas-Boas, and Joao Cardos

I would like to show my gratitude to Jana Seidel (Project Manager) for her help and support.

This work will not be achieved well, without their support.

Thank you so much

1. Research objective
   This survey evaluates our research approach. Our research aims to build a re-engineering approach to support the transformation of a products family into systematic reuse approach, most likely an SPL. To transform existing products and build a systematic reuse base approach, an FM must be derived as an initial step. For this reason, our approach (1) obtains a systematic domain abstraction for the commonality and variability information of existing products in a variability model (i.e., an FM), and then it (2) maps the commonality and variability information (features) to the code. At the same time, our approach (3) evolves a products family with a new product; moreover, it derives a product from a products family using the specifications of the new product.

2. Target population
   The target population of our case study are the developers that work with Classical Sensor Variants Family software development team at Bosch Car Multimedia (the Bosch company at Braga).

3. Sampling design
   It will be given to the Classical Sensor Variants Family software development department and the developers will be asked to answer a questionnaire and to attend a personal meeting.

4. The questionnaire
   Please answer the questions as completely as honest as possible. Try to answer every question. In case, you are not sure about the answer, just select the one you feel is the most likely. Thank you so much for your support.

B.1 Background information

1.	What is your major field of study?						
2.	How much experience do you have in your work?						
a)	Little experience	b)	Medium experience	c)	Significant experience	d)	Professional experience
3.	How motivated are you to perform well in this experience?						
a)	Highly unmotivated	b)	Unmotivated	c)	Motivated	d)	Highly motivated
4.	What is your position in the department?						

B.2 The experiment

You take the role of a software developer who investigates a set of products of the Classical Sensor Variants Family for commonality and variability. Please use the output documents of our approach to perform the experiment.

1. The current FM.
2. Traceability tree.

Please perform the experiment three times at least and answer the following questions:

Write down the start time here (e.g., 13:30) ——————-

Write down the end time here (e.g., 14:30) ——————-

To what degree do you agree with the following statements?

1.	Is it possible to have a Counterclockwise and Clockwise options for the Rotation sense feature at the same time in the product implementation?				
a)	Yes	b)	No	c)	I don't Know
2.	Product 154 uses algorithm1 to indicate a checksum for CHK_SUM feature. Do you agree with this statement?				
a)	Yes	b)	No	c)	I don't Know
3.	To calibrate the steering angle sensor, Product 154 use the command code word with the value(s):				
a)	CCWB2	b)	CCW23	d)	I don't Know
4.	Beyond evolving the family with a new features, please list down the new features:				
5.	Write down the routine name to activate the functionality of the feature LWS_Identification in a produ				
6.	Write down the routine name to update the code of the new product with the required implementation of the new feature LWS_SPD_VD:				
7.	To evolve the family with a new product. Is it required including the feature in its code implementation				
a)	Yes	d)	No	e)	I don't Know
8.	The available set of products have a high degree of commonality.				
a)	Yes	b)	No	c)	I don't know

Appendix C

EVoSPL feedback

C.1 Developers feedback

1.	Using the current FM and traceability tree could reduce the systematic reuse transformation effort.						
a)	Strongly disagree	b)	Disagree	c)	Agree	d)	Strongly agree
2.	Using the current FM and traceability tree could increase the degree of reuse achieved in the systematic reuse transformation.						
a)	Strongly disagree	b)	Disagree	c)	Agree	d)	Strongly agree
3.	In case the systematic transformation reuse is not attempted, our approach could reduce the effort or further and future parallel maintenance of a products family.						
a)	Strongly disagree	b)	Disagree	c)	Agree	d)	Strongly agree
4.	Using the current FM and the traceability tree could reduce the effort of evolving the products family with a new product.						
a)	Strongly disagree	b)	Disagree	c)	Agree	d)	Strongly agree
5.	Using the current FM increases the understandability of a products family domain.						
a)	Strongly disagree	b)	Disagree	c)	Agree	d)	Strongly agree
6.	Using the traceability tree allows the developer to find the implemented features within a code with a higher degree of correctness						
a)	Strongly disagree	b)	Disagree	c)	Agree	d)	Strongly agree
7.	Using the current FM, clearly, I can determine the common functionalities among a products family.						
a)	Strongly disagree	b)	Disagree	c)	Agree	d)	Strongly agree
8.	I think the output of the research approach provides good support for solving the tasks correctly.						
a)	Strongly disagree	b)	Disagree	c)	Agree	d)	Strongly agree
9.	Using the current FM, I could easily see the variability between the products of a family and relates them to their implementation in the code.						
a)	Strongly disagree	b)	Disagree	c)	Agree	d)	Strongly agree
10.	I recommended the approach to be used for systematic reuse, almost possibly an SPL.						
a)	Strongly disagree	b)	Disagree	c)	Agree	d)	Strongly agree

C.2 Comparison to baseline approach

1.	Using the research approach could reduce the systematic reuse transformation effort over the baseline approach.						
a)	Strongly disagree	b)	Disagree	c)	Agree	d)	Strongly agree
2.	Using the current FM and the traceability tree could increase the degree of reuse achieved in the systematic reuse transformation over the baseline approach.						
a)	Strongly disagree	b)	Disagree	c)	Agree	d)	Strongly agree
3.	In case the systematic transformation reuse is not attempted, our approach could reduce the effort for further and future parallel maintenance of a products family over the baseline approach.						
a)	Strongly disagree	b)	Disagree	c)	Agree	d)	Strongly agree
4.	Using the current FM and the traceability tree could reduce the effort of evolving the products family with a new product over the baseline approach.						
a)	Strongly disagree	b)	Disagree	c)	Agree	d)	Strongly agree
5.	Using the current FM increases the understandability of a products family domain over the baseline approach.						
a)	Strongly disagree	b)	Disagree	c)	Agree	d)	Strongly agree
6.	Using the traceability tree allows the developer to find the implemented features within the code with a higher degree of correctness over the baseline approach.						
a)	Strongly disagree	b)	Disagree	c)	Agree	d)	Strongly agree
7.	Using the current FM, clearly, I can determine the common functionalities among a products family over the baseline approach.						
a)	Strongly disagree	b)	Disagree	c)	Agree	d)	Strongly agree
8.	I could easily see the variability among the products of a family and relate them to their implementatio in the code over the baseline approach.						
a)	Strongly disagree	b)	Disagree	c)	Agree	d)	Strongly agree

Bibliography

[Ach+11] Mathieu Acher et al. “Reverse engineering architectural feature models”. In: European Conference on Software Architecture. Springer. 2011, pp. 220–235. DOI: 10.1007/978-3-642-23798-0_25.

[Ach+12] Mathieu Acher et al. “On extracting feature models from product descriptions”. In: Sixth International Workshop on Variability Modeling of Software-Intensive Systems. 2012, pp. 45–54.

[AH06] Richard Allen and Charles Hulme. “Speech and language processing mechanisms in verbal serial recall”. In: Journal of Memory and Language 55.1 (2006), pp. 64–88.

[Alf+08] Mauricio Alférez et al. “A Model-driven Approach for Software Product Lines Requirements Engineering.” In: SEKE. 2008, pp. 779–784.

[Alv+05] Vander Alves et al. “Extracting and evolving mobile games product lines”. In: International Conference on Software Product Lines. Springer. 2005, pp. 70–81.

[Alv+06a] Vander Alves et al. “Bidirectional refactorings: Catalog and completeness”. In: TR-UFPE-CIN-200608028. Federal University of Pernambuco. 2006, pp. 111–128.

[Alv+06b] Vander Alves et al. “Refactoring product lines”. In: Fifth international conference on Generative programming and component engineering. 2006, pp. 201–210. DOI: 10.1145/1173706.1173737.

[Alv+10] Vander Alves et al. “Requirements engineering for software product lines: A systematic literature review”. In: Information and Software Technology 52.8 (2010), pp. 806–820. DOI: 10.1016/j.infsof.2010.03.014.

[AM14] R Al-Msie’deen. “Reverse engineering feature models from software variants to build software product lines: REVPLINE approach”. In: Theses, Universite Montpellier II-Sciences et Techniques du Languedoc (2014).

[AOH04] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. “A differencing algorithm for object-oriented programs”. In: 19th International Conference on Automated Software Engineering, 2004. IEEE. 2004, pp. 2–13.

[Ape+13] Sven Apel et al. “A development process for feature-oriented product lines”. In: Feature-Oriented Software Product Lines. Springer, 2013, pp. 17–44.

[Ape+18] S Apel et al. “Feature-oriented software product lines: Concepts and implementation”. In: URL http://www. springer. com/computer/swe/book/978-3-642-37520-0 (2018).

[AR12] Hugo Arboleda and Jean-Claude Royer. Model-driven and software product line engineering. Wiley Online Library, 2012.

[Ass15] Wesley Klewerton Guez Assunção. “Search-based migration of model variants to software product line architectures”. In: 37th IEEE International Conference on Software Engineering. Vol. 2. IEEE. 2015, pp. 895–898.

[Ass+17] Wesley KG Assunção et al. “Reengineering legacy applications into software product lines: a systematic mapping”. In: Empirical Software Engineering 22.6 (2017), pp. 2972–3016.

[AT02] Samuel A Ajila and Patrick J Tierney. “The foom method-modeling software product lines in industrial settings”. In: International Conference on Software Engineering Research and Practice (SERP02). Las Vegas, Nevada, USA: CSREA Press. 2002.

[AW05] S Massoud Amin and Bruce F Wollenberg. “Toward a smart grid: power delivery for the 21st century”. In: IEEE power and energy magazine 3.5 (2005), pp. 34–41.

[Bay+99] Joachim Bayer et al. “PuLSE: A methodology to develop software product lines”. In: symposium on Software reusability. 1999, pp. 122–131.

[BBM05] Kathrin Berg, Judith Bishop, and Dirk Muthig. “Tracing software product line variability: from problem to solution space”. In: SAICSIT. Vol. 5. Citeseer. 2005, pp. 182–191.

[BCH15] Jan Bosch, Rafael Capilla, and Rich Hilliard. “Trends in Systems and Software Variability [Guest editors’ introduction]”. In: IEEE Software 32.3 (2015), pp. 44–51.

[Ber+13] Thorsten Berger et al. “A survey of variability modeling in industrial practice”. In: Seventh International Workshop on Variability Modelling of Software-intensive Systems. 2013, pp. 1–8.

[Beu12] Danilo Beuche. “Modeling and building software product lines with pure:: variants”. In: 16th International Software Product Line Conference-Volume 2. 2012, pp. 255–255.

[Beu15] Danilo Beuche. “Managing Variability with Feature Models”. In: 15th International Software Product Lines Conference. 2015.

[Big] BigLever. accessed September 6, 2020. URL: https://biglever.com/solution/gears/.

[BKS15] Noor Hasrina Bakar, Zarinah M Kasirun, and Norsaremah Salleh. “Feature extraction approaches from natural language requirements for reuse in software product lines: A systematic literature review”. In: Journal of Systems and Software 106 (2015), pp. 132–149. DOI: 10.1016/j.jss.2015.05.006.

[BLL08] Hongyu Pei Breivold, Stig Larsson, and Rikard Land. “Migrating industrial systems towards software product lines: Experiences and observations through case studies”. In: 34th Euromicro Conference Software Engineering and Advanced Applications. IEEE. 2008, pp. 232–239.

[BM07] Alexandre Braganca and Ricardo J Machado. “Automating mappings between use case diagrams and feature models for software product lines”. In: Eleventh international software product line conference (SPLC 2007). IEEE. 2007, pp. 3–12. DOI: 10.1109/SPLINE.2007.17.

[Bor09] Paulo Borba. “An introduction to software product line refactoring”. In: International Summer School on Generative and Transformational Techniques in Software Engineering. Springer. 2009, pp. 1–26. DOI: 10.1007/978-3-642-18023-1_1.

[Bos] Bosch em Portugal. accessed July 6, 2020. URL: https://www.bosch.pt/.

[Bos00] Jan Bosch. Design and use of software architectures: adopting and evolving a product-line approach. Pearson Education, 2000.

[Bos02] Jan Bosch. “Maturity and evolution in software product lines: Approaches, artefacts and organization”. In: International Conference on Software Product Lines. Springer. 2002, pp. 257–271.

[Bot+10] Goetz Botterweck et al. “EvoFM: feature-driven planning of product-line evolution”. In: ICSE Workshop on Product Line Approaches in Software Engineering. 2010, pp. 24–31.

[Bot+11] Goetz Botterweck et al. “Towards feature-driven planning of product-line evolution”. In: First International Workshop on Feature-Oriented Software Development. 10.1145/1629716.1629737, pp. 109–116.

[BP14] Goetz Botterweck and Andreas Pleuss. “Evolution of software product lines”. In: Evolving Software Systems. Springer, 2014, pp. 265–295. DOI: 10.1007/978-3-642-45398-4_9.

[BPSP04] Danilo Beuche, Holger Papajewski, and Wolfgang Schröder-Preikschat. “Variability management with feature models”. In: Science of Computer Programming 53.3 (2004), pp. 333–352. DOI: 10.1016/j.scico.2003.04.005.

[Bra+14] Peter Braun et al. “Guiding requirements engineering for software-intensive embedded systems in the automotive industry”. In: Computer Science-Research and Development 29.1 (2014), pp. 21–43. DOI: 10.1007/s00450-010-0136-y.

[Bro+09] Manfred Broy et al. “Automotive architecture framework: Towards a holistic and standardised system architecture description”. In: IBM Corporation and TUM Technical Report. Semantic Scholar. 2009.

[Cal+07] Fernando Calheiros et al. “Product Line Variability Refactoring Tool.” In: WRT. 2007, pp. 32–33.

[CC90] Elliot J. Chikofsky and James H Cross. “Reverse engineering and design recovery: A taxonomy”. In: IEEE software 7.1 (1990), pp. 13–17.

[CD12] Hernán Casalánguida and Juan Eduardo Durán. “Automatic generation of feature models from UML requirement models”. In: 16th International Software Product Line Conference. 2012, pp. 10–17.

[CH+14] Jane Cleland-Huang et al. “Software traceability: trends and future directions”. In: Future of Software Engineering Proceedings. 2014, pp. 55–69.

[CHGZ+12] Jane Cleland-Huang, Orlena Gotel, Andrea Zisman, et al. Software and systems traceability. Vol. 2. 3. Springer, 2012.

[Chi06] Phan Cong Chinh. “An approach to adaptive inference engine for rule-based consultation systems”. PhD thesis. 2006.

[CN01] PC Clements and L Northrop. “Software Product Lines: Practices and Patterns. SEI Ser”. In: SE. Addison-Wesley (2001).

[Cor] CoreNPL. accessed July 1, 2020. URL: https://stanfordnlp.github.io/CoreNLP/.

[CVF11] Marcus Vinicius Couto, Marco Tulio Valente, and Eduardo Figueiredo. "Extracting software product lines: A case study using conditional compilation". In: 15th European Conference on Software Maintenance and Reengineering. IEEE. 2011, pp. 191–200.

[Dav+13] Jean-Marc Davril et al. "Feature model extraction from large collections of informal product descriptions". In: Ninth Joint Meeting on Foundations of Software Engineering. 2013, pp. 290–300. DOI: 10.1145/2491411.2491455.

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Object-oriented reengineering patterns. Elsevier, 2002.

[Dhu+08] Deepak Dhungana et al. "Supporting evolution in model-based product line engineering". In: 12th International Software Product Line Conference. IEEE. 2008, pp. 319–328.

[Difa] DiffNow. accessed July 1, 2020. URL: https://www.diffchecker.com/.

[Difb] DiffNow. accessed July 1, 2020. URL: https://www.diffnow.com/compareclips.

[Dit+13] Bogdan Dit et al. "Feature location in source code: a taxonomy and survey". In: Journal of software: Evolution and Process 25.1 (2013), pp. 53–95.

[DJT01] Xuehong Du, Jianxin Jiao, and Mitchell M Tseng. "Architecture of product family: fundamentals and methodology". In: Concurrent Engineering 9.4 (2001), pp. 309–325.

[DL18] Li Deng and Yang Liu. Deep learning in natural language processing. Springer, 2018.

[DPG15] Jessica Díaz, Jennifer Pérez, and Juan Garbajosa. "A model for tracing variability from features to product-line architectures: a case study in smart grids". In: Requirements Engineering 20.3 (2015), pp. 323–343.

[Dra] Draftable. accessed July 6, 2020. URL: https://draftable.com/.

[Dub+13] Yael Dubinsky et al. "An exploratory study of cloning in industrial software product lines". In: 17th European Conference on Software Maintenance and Reengineering. IEEE. 2013, pp. 25–34. DOI: 10.1109/CSMR.2013.13.

[Dus15] Slawomir Duszynski. Analyzing similarity of cloned software variants using hierarchical set models. Fraunhofer IRB Verlag, 2015. DOI: 10.5555/2809180.

[EBB04] Magnus Eriksson, Jürgen Börstler, and Kjell Borg. "Marrying features and use cases for product line requirements modeling of embedded systems". In: Fourth Conference on Software Engineering Research and Practice in Sweden SERPS. Vol. 4. 2004, pp. 73–82.

[EBB05] Magnus Eriksson, Jürgen Börstler, and Kjell Borg. "The PLUSS approach–domain modeling with features, use cases and use case realizations". In: International Conference on Software Product Lines. Springer. 2005, pp. 33–44.

[Ecl] Eclipse Foundation. accessed July 1, 2020. URL: https://www.eclipse.org/ide/.

[ES+12] Hamzeh Eyal-Salman et al. “Genetic Algorithms as a Recovering Traceability Links Method between Feature Models and Source Code of Product Variants”. In: Actes de la Journée Lignes de Produits (2012), pp. 3–14.

[ESSD14] Hamzeh Eyal-Salman, Abdelhak-Djamel Seriai, and Christophe Dony. “Feature location in a collection of product variants: Combining information retrieval and hierarchical clustering”. In: SEKE: Software Engineering and Knowledge Engineering. 2014, pp. 426–430.

[Fen+17] Wolfram Fenske et al. “Variant-preserving refactorings for migrating cloned products to a product line”. In: 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE. 2017, pp. 316–326.

[Fis+14] Stefan Fischer et al. “Enhancing clone-and-own with systematic reuse for developing software variants”. In: International Conference on Software Maintenance and Evolution. IEEE. 2014, pp. 391–400.

[Fly06] Bent Flyvbjerg. “Five misunderstandings about case-study research”. In: Qualitative inquiry 12.2 (2006), pp. 219–245.

[FM16] João M Fernandes and Ricardo J Machado. Requirements in engineering projects. Springer, 2016.

[Fri] friendlyMapper. accessed September 15, 2020. URL: https://github.com/it-karam/friendlyMapper.

[Gal+11] Matthias Galster et al. “Variability in software architecture: current practice and challenges”. In: ACM SIGSOFT Software Engineering Notes 36.5 (2011), pp. 30–32.

[Gar+03] Akash Garg et al. “An environment for managing evolving product line architectures”. In: International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings. IEEE. 2003, pp. 358–367.

[GF13] Nadia Gamez and Lidia Fuentes. “Architectural evolution of FamiWare using cardinality-based feature models”. In: Information and Software Technology 55.3 (2013), pp. 563–580.

[GF94] Orlena CZ Gotel and CW Finkelstein. “An analysis of the requirements traceability problem”. In: International Conference on Requirements Engineering. IEEE. 1994, pp. 94–101.

[GFd98] Martin L Griss, John Favaro, and Massimo d’Alessandro. “Integrating feature modeling with the RSEB”. In: Fifth International Conference on Software Reuse (Cat. No. 98TB100203). IEEE. 1998, pp. 76–85.

[Ghe+06] Rohit Gheyi et al. Theory and proofs for feature model refactorings. Tech. rep. Technical Report TR-UFPE-CIN-200608027, Federal University of Pernambuco, 2006.

[Git] Why Git Hub. accessed July 6, 2020. URL: https://github.com/.

[GM09] Yaser Ghanam and Frank Maurer. “Extreme product line engineering: Managing variability and traceability via executable specifications”. In: Agile Conference. IEEE. 2009, pp. 41–48. DOI: 10.1109/AGILE.2009.12.

[Gom05] Hassan Gomaa. “Designing Software Product Lines with UML.” In: SEW Tutorial Notes. 2005, pp. 160–216.

[Gom06] Hassan Gomaa. “A software modeling odyssey: designing evolutionary architecture-centric real-time systems and product lines”. In: International Conference on Model Driven Engineering Languages and Systems. Springer. 2006, pp. 1–15.

[Gom11] Hassan Gomaa. Software modeling and design: UML, use cases, patterns, and software architectures. Cambridge University Press, 2011.

[Gom13] Hassan Gomaa. “Evolving software requirements and architectures using software product line concepts”. In: Second International Workshop on the Twin Peaks of Requirements and Architecture (TwinPeaks). IEEE. 2013, pp. 24–28.

[Guo+11] Jianmei Guo et al. “A genetic algorithm for optimized feature selection with resource constraints in software product lines”. In: Journal of Systems and Software 84.12 (2011), pp. 2208–2221. DOI: 10.1016/j.jss.2011.06.026.

[Hin+18] Daniel Hinterreiter et al. “Feature-oriented evolution of automation software systems in industrial software ecosystems”. In: 23rd International Conference on Emerging Technologies and Factory Automation (ETFA). Vol. 1. IEEE. 2018, pp. 107–114.

[HTM09] Herman Hartmann, Tim Trew, and Aart AJ Matsinger. “Supplier independent feature modelling.” In: SPLC. Vol. 9. 2009, pp. 191–200.

[ID10] Nitin Indurkhya and Fred J Damerau. Handbook of natural language processing. Vol. 2. CRC Press, 2010.

[IF19] Karam Ignaim and João M Fernandes. “An Industrial Case Study for Adopting Software Product Lines in Automotive Industry: An Evolution-Based Approach for Software Product Lines (EVOA-SPL)”. In: 23rd International Systems and Software Product Line Conference-Volume B. 2019, pp. 183–190.

[IRB14] Nili Itzik and Iris Reinhartz-Berger. “Generating feature models from requirements: Structural vs. functional perspectives”. In: 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools-Volume 2. 2014, pp. 44–51.

[IRBW15] Nili Itzik, Iris Reinhartz-Berger, and Yair Wand. “Variability analysis of requirements: Considering behavioral differences and reflecting stakeholders’ perspectives”. In: IEEE Transactions on Software Engineering 42.7 (2015), pp. 687–706. DOI: 10.1016/j.scico.2012.

[JB08] Mikoláš Janota and Goetz Botterweck. “Formal approach to integrating feature and architecture models”. In: International Conference on Fundamental Approaches to Software Engineering. Springer. 2008, pp. 31–45.

[JT00] Jianxin Jiao and Mitchell M Tseng. “Understanding product family for mass customization by developing commonality indices”. In: Journal of Engineering Design 11.3 (2000), pp. 225–243.

[KA11] Christian Kästner and Sven Apel. “Feature-oriented software development”. In: International Summer School on Generative and Transformational Techniques in Software Engineering. Springer. 2011, pp. 346–382.

[KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. “Granularity in software product lines”. In: 30th International Conference on Software Engineering. IEEE. 2008, pp. 311–320.

[Kan+90] Kyo C Kang et al. Feature-oriented domain analysis (FODA) feasibility study. Tech. rep. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[Kas+09] Christian Kastner et al. “FeatureIDE: A tool framework for feature-oriented software development”. In: 31st International Conference on Software Engineering. IEEE. 2009, pp. 611–614.

[Ken] Kent State. accessed July 13, 2020. URL: https://libguides.library.kent.edu/SPSS/PairedSamplestTest.

[Kit96] Barbara Ann Kitchenham. “Evaluating software engineering methods and tool part 1: The evaluation context and evaluation methods”. In: ACM SIGSOFT Software Engineering Notes 21.1 (1996), pp. 11–14.

[KP98] Barbara Ann Kitchenham and Lesley M Pickard. “Evaluating software engineering methods and tools: part 9: quantitative case study methodology”. In: ACM SIGSOFT Software Engineering Notes 23.1 (1998), pp. 24–26.

[Kru01] CharlesW Krueger. “Easing the transition to software mass customization”. In: International Workshop on Software Product-Family Engineering. Springer. 2001, pp. 282–293. DOI: 10.1007/3-540-47833-7_25.

[Kru92] Charles W Krueger. “Software reuse”. In: ACM Computing Surveys (CSUR) 24.2 (1992), pp. 131–183.

[KS98] Sridhar Kota and Kannan Sethuraman. “Managing variety in product families through design for commonality”. In: ASME Design Engineering Technical Conferences, Atlanta, Georgia. DTM-5651. 1998.

[KSRB13] Dean Kramer, Christian Sauer, and Thomas Roth-Berghofer. “Towards explanation generation using feature models in software product lines”. In: Knowledge Engineering and Software Engineering (KESE) (2013), p. 13.

[Kum+18] A Charan Kumari et al. “Feature selection optimization in SPL using genetic algorithm”. In: Procedia Computer Science 132 (2018), pp. 1477–1486.

[LC13] Miguel A Laguna and Yania Crespo. “A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring”. In: Science of Computer Programming 78.8 (2013), pp. 1010–1034.

[Li+16] Li Li et al. “Mining families of android applications for extractive SPL adoption”. In: 20th International Systems and Software Product Line Conference. 2016, pp. 271–275.

[Lie+15] Jörg Liebig et al. “Morpheus: Variability-aware refactoring in the wild”. In: 37th IEEE International Conference on Software Engineering. Vol. 1. IEEE. 2015, pp. 380–391. DOI: 10.1109/ICSE.2015.57.

[Lin+15] Johan Linaker et al. “Guidelines for conducting surveys in software engineering v. 1.1”. In: Lund University (2015).

[Liu+07] Jing Liu et al. “State-based modeling to support the evolution and maintenance of safety-critical software product lines”. In: 14th Annual IEEE

International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07). IEEE. 2007, pp. 596–608.

[Liv11] Steve Livengood. "Issues in software product line evolution: complex changes in variability models". In: Second International Workshop on Product Line Approaches in Software Engineering. 2011, pp. 6–9. DOI: 10.1145/1985484.1985487.

[LSR07] Frank van der Linden, Klaus Schmid, and Eelco Rommes. "The product line engineering approach". In: Software Product Lines in Action. Springer, 2007, pp. 3–20.

[LSS05] Timothy C Lethbridge, Susan Elliott Sim, and Janice Singer. "Studying software engineers: Data collection techniques for software field studies". In: Empirical software engineering 10.3 (2005), pp. 311–341.

[LSS17] Yang Li, Sandro Schulze, and Gunter Saake. "Reverse engineering variability from natural language documents: A systematic literature review". In: 21st International Systems and Software Product Line Conference-Volume A. 2017, pp. 133–142.

[Mac+14] Lucas Machado et al. "Splconfig: Product configuration in software product line". In: Brazilian Congress on Software (CBSoft), Tools Session. 2014, pp. 1–8.

[Mar+14] Jabier Martinez et al. "Identifying and visualising commonality and variability in model variants". In: European Conference on Modelling Foundations and Applications. Springer. 2014, pp. 117–131.

[Mar+15a] Jabier Martinez et al. "Automating the extraction of model-based software product lines from model variants (T)". In: 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE. 2015, pp. 396–406.

[Mar+15b] Jabier Martinez et al. "Bottom-up adoption of software product lines: a generic and extensible approach". In: 19th International Conference on Software Product Line. 2015, pp. 101–110.

[Mar+16a] Jabier Martinez et al. "Feature location benchmark for software families using eclipse community releases". In: International Conference on Software Reuse. Springer. 2016, pp. 267–283.

[Mar+16b] Jabier Martinez et al. "Name suggestions during feature identification: the variclouds approach". In: 20th International Systems and Software Product Line Conference. 2016, pp. 119–123.

[Mar+17] Jabier Martinez et al. "Bottom-up technologies for reuse: automated extractive adoption of software product lines". In: 39th International Conference on Software Engineering Companion (ICSE-C). IEEE. 2017, pp. 67–70.

[Mar+19] Maira Marques et al. "Software product line evolution: A systematic literature review". In: Information and Software Technology 105 (2019), pp. 190–208. DOI: 10.1016/j.infsof.2018.08.014.

[MD16] Leticia Montalvillo and Oscar Díaz. "Requirement-driven evolution in software product lines: A systematic mapping study". In: Journal of Systems and Software 122 (2016), pp. 110–143.

[Mei+16] Jens Meinicke et al. “FeatureIDE: taming the preprocessor wilderness”. In: 38th International Conference on Software Engineering Companion (ICSE-C). IEEE. 2016, pp. 629–632.

[Mei+17] Jens Meinicke et al. “Feature Traceability in Feature Models and Configurations”. In: Mastering Software Variability with FeatureIDE. Springer, 2017, pp. 73–80. DOI: 10.1007/978-3-319-61443-4_7.

[Men+09] Cem Mengi et al. “Model-driven Support for Source Code Variability in Automotive Software Engineering”. In: First MAPLE Workshop. 2009, pp. 44–50.

[Met+07] Andreas Metzger et al. “Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis”. In: 15th IEEE International Requirements Engineering Conference (RE 2007). IEEE. 2007, pp. 243–253.

[ML04] Thomas von der Maßen and Horst Lichter. “Deficiencies in feature models”. In: workshop on software variability management for product derivation-towards tool support. Vol. 44. 2004, p. 21.

[MP07] Andreas Metzger and Klaus Pohl. “Variability management in software product line engineering”. In: 29th International Conference on Software Engineering (ICSE’07 Companion). IEEE. 2007, pp. 186–187.

[MP14] Andreas Metzger and Klaus Pohl. “Software product line engineering and variability management: achievements and challenges”. In: Future of Software Engineering Proceedings. 2014, pp. 70–84. DOI: 10.1145/2593882.2593888.

[MRR04] Jürgen Meister, Ralf Reussner, and Martin Rohde. “Managing product line variability by patterns”. In: Net. ObjectDays: International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World. Springer. 2004, pp. 153–168.

[MSC14] Tom Mens, Alexander Serebrenik, and Anthony Cleve. Evolving Software Systems. Vol. 190. Springer, 2014.

[Mus+18] Karam Mustafa et al. “A Systematic Reuse-Based Approach for Customized Cloned Variants”. In: Eleventh International Conference on the Quality of Information and Communications Technology (QUATIC). IEEE. 2018, pp. 287–292.

[Nai13] Divya Karunakaran Nair. “Variability-Modelling Practices in Industrial Software Product Lines: A Qualitative Study”. MA thesis. University of Waterloo, 2013.

[Nev+11] Laís Neves et al. “Investigating the safe evolution of software product lines”. In: 10th ACM international conference on Generative programming and component engineering. 2011, pp. 33–42.

[NM11] Linus Nyman and Tommi Mikkonen. “To fork or not to fork: Fork motivations in SourceForge projects”. In: International Journal of Open Source Software and Processes (IJOSSP) 3.3 (2011), pp. 1–9.

[OA15] Raphael Pereira de Oliveira and Eduardo Santana de Almeida. “Requirements evolution in software product lines: An empirical study”. In: IX

Brazilian Symposium on Components, Architectures and Reuse Software. IEEE. 2015, pp. 1–10.

[Pas+10] Leonardo Passos et al. “Coevolution of variability models and related software artifacts”. In: Empirical Software Engineering 21.4 (10.1007/s10664-015-9364-x), pp. 1744–1793.

[PBDL05] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. Software product line engineering: foundations, principles and techniques. Springer Science & Business Media, 2005.

[Ple+12] Andreas Pleuss et al. “Model-driven support for product line evolution on feature level”. In: Journal of Systems and Software 85.10 (2012), pp. 2261–2274.

[Pur] pureVariant. accessed July 6, 2020. URL: http://www.pure-systems.com/pure_variants.49.0.html.

[Rab+08] Daniela Rabiser et al. “Value-based elicitation of product line variability: An experience report”. In: Second International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS). Citeseer, 2008, pp. 73–79.

[Rab+10] Rick Rabiser et al. “A flexible approach for generating product-specific documents in product lines”. In: International Conference on Software Product Lines. Springer. 2010, pp. 47–61.

[Rab+16] Daniela Rabiser et al. “A prototype-based approach for managing clones in clone-and-own product lines”. In: 20th International Systems and Software Product Line Conference. 2016, pp. 35–44. DOI: 10.1145/2934466.2934487.

[Ra 3] AL Ra’Fat et al. “Feature location in a collection of software product variants using formal concept analysis”. In: International Conference on Software Reuse. Springer. 2013, pp. 302–307.

[RCC13] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. “Managing cloned variants: a framework and experience”. In: 17th International Software Product Line Conference. 2013, pp. 101–110.

[RDR03] Claudio Riva and Christian Del Rosso. “Experiences with software product family evolution”. In: Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings. IEEE. 2003, pp. 161–169.

[RE14] Olivier Renault and PLM Industry Expert. “Reuse/Variability Management and System Engineering”. In: Poster Workshop of the Complex Systems Design & Management Conference CSD&M 2014. Vol. 173. Citeseer. 2014.

[RH09] Per Runeson and Martin Höst. “Guidelines for conducting and reporting case study research in software engineering”. In: Empirical software engineering 14.2 (2009), p. 131.

[RJN16] Ashwini Rupnawar, Ashwini Jagdale, and Samiksha Navsupe. “Study on Forward Chaining and Reverse Chaining in Expert System”. In: International Journal of Advanced Engineering Research and Science 3.12 (2016), p. 236945.

[Rom+13] Daniel Romero et al. “SPLEMMA: A generic framework for controlled-evolution of software product lines”. In: 17th International Software Product Line Conference co-located workshops. 2013, pp. 59–66.

[Rub+12] Julia Rubin et al. “Managing forked product variants”. In: 16th International Software Product Line Conference-Volume 1. 2012, pp. 156–160.

[RW05] Mark Oliver Reiser and Matthias Weber. “Using product sets to define complex product decisions”. In: International Conference on Software Product Lines. Springer. 2005, pp. 21–32.

[San] Sanfoundry. accessed May 1, 2019. URL: https://www.sanfoundry.com/c-program-atm-transaction/.

[SB99] Mikael Svahnberg and Jan Bosch. “Evolution in software product lines: Two cases”. In: Journal of Software Maintenance: Research and Practice 11.6 (1999), pp. 391–422.

[SCC16] Samuel Sepúlveda, Ania Cravero, and Cristina Cachero. “Requirements modeling languages for software product lines: A systematic literature review”. In: Information and Software Technology 69 (2016), pp. 16–36. DOI: 10.1016/j.infsof.2015.08.007.

[Sch+12] Sandro Schulze et al. “Variant-preserving refactoring in feature-oriented software product lines”. In: Sixth International Workshop on Variability Modeling of Software-Intensive Systems. 2012, pp. 73–81.

[Scr] Scribber. accessed July 12, 2020. URL: https://www.scribbr.com/methodology/hypothesis-testing/.

[Ser+13] Abdelhak-Djamel Seriai 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: 25th International Conference on Software Engineering and Knowledge Engineering. Knowledge Systems Institute Graduate School. 2013, p. 8.

[SHA12] Christoph Seidl, Florian Heidenreich, and Uwe Aßmann. “Co-evolution of models and feature mapping in software product lines”. In: 16th International Software Product Line Conference-Volume 1. 2012, pp. 76–85.

[She+11] Steven She et al. “Reverse engineering feature models”. In: 33rd International Conference on Software Engineering. 2011, pp. 461–470.

[Sib+13] Elisa Margareth Sibarani et al. “A study of parsing process on natural language processing in Bahasa Indonesia”. In: 16th International Conference on Computational Science and Engineering. IEEE. 2013, pp. 309–316.

[SKL08] Pieter van der Spek, Steven Klusener, and Pierre van de Laar. “Towards recovering architectural concepts using latent semantic indexing”. In: 12th European Conference on Software Maintenance and Reengineering. IEEE. 2008, pp. 253–257.

[SO01] Christoph Stoermer and Liam O’Brien. “MAP-mining architectures for product line evaluations”. In: Proceedings Working IEEE/IFIP Conference on Software Architecture. IEEE. 2001, pp. 35–44.

[SOB02] Dennis Smith, Liam O’Brien, and John Bergey. “Using the options analysis for reengineering (OAR) method for mining components for a product line”.

In: International Conference on Software Product Lines. Springer. 2002, pp. 316–327.

[SSA14] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. “Capturing variability in space and time with hyper feature models”. In: Eighth International Workshop on Variability Modelling of Software-Intensive Systems. 2014, pp. 1–8.

[SSW09] Ştefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. “Variability management in software product lines: a systematic review”. In: 13th International Software Product Line Conference. Association for Computing Machinery, 2009.

[SSW15] Ştefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. “Forked and integrated variants in an open-source firmware project”. In: International Conference on Software Maintenance and Evolution (ICSME). IEEE. 2015, pp. 151–160.

[Sta] Statistics Solutions. accessed July 3, 2020. URL: https://www.statisticssolutions.com/manova-analysis-paired-sample-t-test/.

[Tan+10] Antony Tang et al. “SPL migration tensions: an industry experience”. In: Workshop on Knowledge-Oriented Product Line Engineering. 2010, pp. 1–6.

[Tel00] Virginia Teller. “Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition”. In: Computational Linguistics 26.4 (2000), pp. 638–641.

[TGB06] Robert Tairas, Jeff Gray, and Ira Baxter. “Visualization of clone detection results”. In: OOPSLA workshop on eclipse technology eXchange. 2006, pp. 50–54.

[THMH16a] Mohammad Tanhaei, Jafar Habibi, and Seyed-Hassan Mirian-Hosseinabadi. “A feature model based framework for refactoring software product line architecture”. In: Journal of Computer Science and Technology 31.5 (2016), pp. 951–986. DOI: 10.1007/s11390-016-1674-y.

[THMH16b] Mohammad Tanhaei, Jafar Habibi, and Seyed-Hassan Mirian-Hosseinabadi. “Automating feature model refactoring: A model transformation approach”. In: Information and Software Technology 80 (2016), pp. 138–157.

[Thü+14] Thomas Thüm et al. “FeatureIDE: An extensible framework for feature-oriented software development”. In: Science of Computer Programming 79 (2014), pp. 70–85.

[TR13] Federico Tomassetti and Daniel Ratiu. “Extracting variability from C and lifting it to mbeddr”. In: International Workshop on Reverse Variability Engineering. 2013.

[Tuf+15] Michele Tufano et al. “When and why your code starts to smell bad”. In: 37th IEEE International Conference on Software Engineering. Vol. 1. IEEE. 2015, pp. 403–414. DOI: 10.1109/ICSE.2015.59.

[UV] Christelle Urtado and Sylvain Vauttier. “Reverse Engineering Feature Models from Software Configurations using Formal Concept Analysis”. In: CLA 2014 (), p. 95.

[Val+17] Tassio Vale et al. “Software product lines traceability: A systematic mapping study”. In: Information and Software Technology 84 (2017), pp. 1–18.

[Wag14] Christian Wagner. “Model-Driven Software Migration”. In: Model-Driven Software Migration: A Methodology. Springer, 2014, pp. 67–105.

[Wan+09] Bo Wang et al. “A use case based approach to feature models’ construction”. In: 17th IEEE International Requirements Engineering Conference. IEEE. 2009, pp. 121–130. DOI: 10.1109/RE.2009.15.

[Wan+12] Fernando Wanderley et al. “Generating feature model from creative requirements using model driven design”. In: 16th International Software Product Line Conference-Volume 2. 2012, pp. 18–25.

[Wij03] Jan Gerben Wijnstra. “Evolving a product family in a changing context”. In: International Workshop on Software Product-Family Engineering. Springer. 2003, pp. 111–128.

[Wik] wikipedia. accessed July 6, 2020. URL: https://en.wikipedia.org/wiki/AUTOSAR.

[Wu+12] Yijian Wu et al. “Towards understanding requirement evolution in a software product line an industrial case study”. In: First IEEE International Workshop on the Twin Peaks of Requirements and Architecture (TwinPeaks). IEEE. 2012, pp. 7–14.

[WW02] Matthias Weber and Joachim Weisbrod. “Requirements engineering in automotive development-experiences and challenges”. In: Proceedings IEEE Joint International Conference on Requirements Engineering. IEEE. 2002, pp. 331–340.

[XXJ10] Yinxing Xue, Zhenchang Xing, and Stan Jarzabek. “Understanding feature evolution in a family of product variants”. In: 17th Working Conference on Reverse Engineering. IEEE. 2010, pp. 109–118.

[XXJ12] Yinxing Xue, Zhenchang Xing, and Stan Jarzabek. “Feature location in a collection of product variants”. In: 19th Working Conference on Reverse Engineering. IEEE. 2012, pp. 145–154. DOI: 10.1109/WCRE.2012.24.

[ZB12] Bo Zhang and Martin Becker. “Code-based variability model extraction for software product line improvement”. In: 16th International Software Product Line Conference-Volume 2. 2012, pp. 91–98.

[ZCA17] Yongjie Zheng, Cuong Cu, and Hazeline U Asuncion. “Mapping features to source code through product line architecture: Traceability and conformance”. In: IEEE International Conference on Software Architecture (ICSA). IEEE. 2017, pp. 225–234. DOI: 10.1109/ICSA.2017.13.

[Zha+06] Wei Zhao et al. “SNIAFL: Towards a static noninteractive approach to feature location”. In: ACM Transactions on Software Engineering and Methodology (TOSEM) 15.2 (2006), pp. 195–226. DOI: 10.1145/1131421.1131424.

[Zho+18] Shurui Zhou et al. “Identifying features in forks”. In: 40th International Conference on Software Engineering (ICSE). IEEE. 2018, pp. 105–116.

[Zia+12] Tewfik Ziadi et al. “Feature identification from the source code of product variants”. In: 16th European Conference on Software Maintenance and Reengineering. IEEE. 2012, pp. 417–422.