Master’s Thesis

Conceptual Variability Management in Software Families with Multiple Contributors

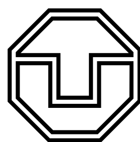
submitted by

David Gollasch

born April 18, 1990 in Altdöbern

Technische Universität Dresden

Fakultät Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie



Supervisors:
Dipl.-Inf. Christoph Seidl
Dr.-Ing. Birgit Demuth
Professor: Prof. Dr. rer. nat. habil. Uwe Aßmann

Submitted November 2, 2015

Abstract

To offer customisable software, there are two main concepts yet: software product lines that allow the product customisation based on a fixed set of variability and software ecosystems, allowing an open product customisation based on a common platform.

Offering a software family that enables external developers to supply software artefacts means to offer a common platform as part of an ecosystem and to sacrifice variability control. Keeping full variability control means to offer a customisable product as a product line, but without the support for external contributors.

This thesis proposes a third concept of variable software: partly open software families. They combine a customisable platform similar to product lines with controlled openness similar to ecosystems.

As a major contribution of this thesis a variability modelling concept is proposed which is part of a variability management for these partly open software families. This modelling concept is based on feature models and extends them to support open variability modelling by means of interfaces, structural interface specifications and the inclusion of semantic information. Additionally, the introduction of a rights management allows multiple contributors to work with the model. This is required to enable external developers to use the model for the concrete extension development.

The feasibility of the proposed model is evaluated using a prototypically developed modelling tool and by means of a case study based on a car infotainment system.

Contents

Abstract **III**

1 Introduction **1**

1.1 Motivation . . . 1
1.2 Research Question and Goals . . . 3
1.3 Thesis Structure . . . 4

2 Partly Open Software Families **5**

2.1 Software Families . . . 5
2.1.1 Variability . . . 6
2.1.2 Software Product Lines . . . 7
2.1.3 Software Ecosystems . . . 10
2.1.4 Commonalities and Differences of SPLs and SECOs . . . 11
2.2 Partly Open Software Families . . . 14
2.2.1 Requirements for the POSF Modelling Concept . . . 17
2.2.2 Chances and Opportunities of POSFs over SPLs and SECOs . . . 19
2.3 Summary . . . 20

3 Analysis of Existing Variability Modelling Methodologies **21**

3.1 Variability Modelling Methodologies . . . 21
3.1.1 Feature Models . . . 22
3.1.2 Decision Models . . . 26
3.1.3 Orthogonal Variability Models . . . 28
3.1.4 Common Variability Language . . . 30
3.1.5 Unique Characteristics of the Presented Modelling Methods . . . 31
3.1.6 Brief Summary of the Presented Modelling Methods . . . 32
3.2 Feasibility Analysis for Partly Open Software Families . . . 33
3.2.1 Feasibility of Feature Models . . . 33
3.2.2 Feasibility of Decision Models . . . 34
3.2.3 Feasibility of Orthogonal Variability Models . . . 36
3.2.4 Feasibility of the Common Variability Language . . . 37
3.2.5 Feasibility Analysis Verdict . . . 39
3.3 Summary . . . 41

4 Creation of a Feasible Concept **43**

4.1 Creation of a Feasible Variability Modelling Concept . . . 43
4.1.1 Introduction of Interface Notation Elements to Feature Models . . . 44
4.1.2 Differentiation of Interface Specifications and Concrete Extensions . . . 45
4.1.3 Addition of Semantic Information to Interface Specifications . . . 46
4.1.4 Introduction of a Rights Management . . . 49
4.1.5 POSF Modelling Requirements Check . . . 51

4.1.6 Model Overview and Example . . . 52
4.2 Development of a Prototypical Tool . . . 53
4.2.1 Conception . . . 53
4.2.2 Implementation . . . 56
4.3 Summary . . . 57

5 Analysis of the Change Impact **61**
5.1 Change Impact Analysis . . . 61
5.1.1 Possible Changes . . . 62
5.1.2 Identification of the Influence Scope . . . 63
5.1.3 Potential Overlap with Other Contributors . . . 67
5.2 Minimisation of Unwanted Impact . . . 68
5.3 Summary . . . 69

6 Evaluation and Case Study **71**
6.1 Conception . . . 71
6.2 Perform the Case Study by Means of the Developed Tool . . . 74
6.2.1 Model the Case Study Construction (Open, Store and Edit Model) . . . 79
6.2.2 Model the Interfaces (Edit Model) . . . 79
6.2.3 Create Users and Assign Features (Edit Model and Open Model as User) 80
6.2.4 Model a Concrete Extension (Edit Model, Store and Import Extension) . 80
6.3 Results and Discussion . . . 81
6.4 Summary . . . 84

7 Conclusion and Outlook **85**
7.1 Conclusion . . . 85
7.1.1 Comparison of SPLs and SECOs . . . 85
7.1.2 Definition of the Concept between SPL and SECO . . . 86
7.1.3 Analysis of Existing Variability Modelling Methodologies . . . 86
7.1.4 Creation of a Feasible Variability Modelling Concept . . . 86
7.1.5 Change Impact Analysis . . . 87
7.2 Outlook . . . 87

A Task Description **i**
A.1 Title . . . i
A.2 Context and Motivation . . . i
A.3 Goals . . . ii
A.3.1 Comparison of SPLs and SECOs . . . ii
A.3.2 Definition of the Concept Between SPL and SECO . . . ii
A.3.3 Analyse Existing Variability Modelling Methodologies . . . ii
A.3.4 Creation of a Feasible Concept . . . ii
A.3.5 Change Impact Analysis . . . iii
A.4 Working Outline . . . iii
A.5 Schedule (Gantt-Chart) . . . iv

B Prototype Manual **ix**
B.1 How to Install and Run the Tool . . . ix
B.2 How to Create a New Model Project . . . xi

B.3 How to Model xii
B.4 How to Store/Export a Model xiii
B.5 How to Open a Model xiii
B.6 How to Import a Partial Model xiv
B.7 How to Apply User Perspectives xiv

C Case Study Construction **xvii**

List of Figures **xxiv**

List of Tables **xxv**

List of Abbreviations **xxvii**

Contents of Electronic Medium (DVD) **xxix**

Bibliography **xxxi**

1 Introduction

1.1 Motivation

Customisable software is an ongoing trend in many fields of computer industry. It combines the advantages of both individually developed software for one specific customer as well as standard commercial off-the-shelf software. Whereas individually ordered and developed software fits the customer's needs best, the monetary effort is relatively high. On the other hand, the price of standard software is low or at least more affordable, but the offered system does not fit the actual customer's requirements ideally as individual software could. Customisable software is a tradeoff in between as it offers to fit the customer's needs better but can still be sold for a reasonable price. [PBL05, p. 13] The advantages reflect onto the vendor's side as the customer base is potentially large, which results in a better Return on Investment (ROI). [BKP04, p. 3]

The trend of customisable software – from a customer's perspective – appears particularly in the field of business software as an alternative to expensive individual software. For instance, SAP offers highly customisable solutions to realise business processes. Business software is mostly related to specific business procedures and workflows, which leads to defined functional and non-functional requirements in that area. In consequence, the demand for highly customisable software results from the necessity of implementing appropriate software – even if the available implementation budget is low. Consumers took over that trend rapidly as they began to demand customised software solutions as well. Smartphones serve as a proving example for this. They offer a very high degree of variability as apps offer the concrete functionality that can be added and removed conveniently. Assuming the release of the first Apple iPhone in 2007 [App07] as the trend's starting point, it is a proof for this trend to see that today in 2015 – only eight years later – regarding the selling numbers, smartphones nearly became a must-have gadget for everyone. For instance, Apple sold 11.63 million iPhones in 2008 and increased this number to 169.22 million in 2014 [App15].

The umbrella term to describe customisable software is *software family*. A software family can be seen as a set of concrete software products that share a reasonable amount of conceptual design and code while offering distinctive differences.

Software Product Lines (SPLs) and Software Ecosystems (SECOs) are two traditional representatives of software families. Both concepts describe software comprising commonalities, which are shared throughout the whole software family, and customisability by allowing variability. Variability of SPLs is offered by the vendor, and the concrete software product can be configured by the customer similar to buying a new car. On the other hand, a SECO offers an open variability approach similar to the apps concept found in smartphones. A SECO usually offers a product platform and allows community-driven supplements. [Bos09]

The relation between both concepts – SPL and SECO – has been topic of numerous publications (cf. [Bos09], [Sch13], [Ber+14]). In most cases, the transition from an SPL to a SECO is discussed and handled as an evolutionary step. These transitions are interpreted as evolving from a closed system to an open one. However, there actually does not seem to exist anything in between both concepts, though a mixture of a closed and open variability approach makes a

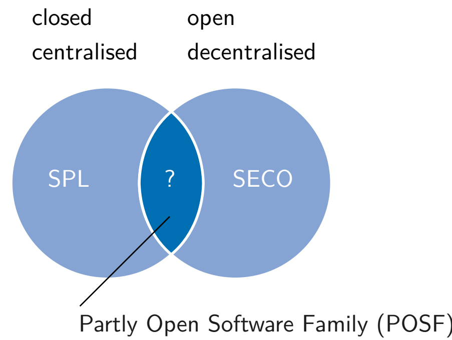


Figure 1.1 – Venn diagram visualisation of SPLs, SECOs and POSFs

promising first impression. A general visualisation is given by the Venn diagram of Figure 1.1 as it shows a combination of "closed world" and "open world" characteristics.

Software families that are a mixture of an open software family similar to a SECO and a closed software family comparable to an SPL can be considered as partly open. Therefore, these kinds of software families will be called Partly Open Software Families (POSFs) in the following. A concrete definition is given in Chapter 2.

The promising impression that arises regarding POSFs are motivated by some particular advantages, which are anticipated to result in designing a family of software products as partly closed and partly open.

- The POSF concept shall allow a seamless collaborative development.
- The development process can be more flexible as it supports closed parts of a system that are under complete control of one developer/software vendor, and it supports open parts to enable others to contribute to the system.
- The software value is supposed to increase as customers get an individually configured product (as known from SPLs) but with even higher flexibility (as known from SECOs). This could result in better sales rates, too.

> *Basically, a POSF allows to open up a lineup of software products to external developers while keeping full control over the system.*

The major problem when creating a POSF is the not yet existing feasible variability management methodology including an appropriate variability modelling concept. Such a modelling concept basically

- has to respect the existence of multiple contributors,
- should consider the change impact caused by these different contributors and
- supports some kind of rights management to define the closed and open parts of a software family.

As preliminary findings and concepts in this research field do not offer a feasible solution here, it shall be the major goal of this thesis to propose a solution.

1.2 Research Question and Goals

The discussed research question of this thesis is the following:

> *What variability modelling concept can be used to allow variability management of Partly Open Software Families as a concept placed between Software Product Lines and Software Ecosystems?*

To answer this question, it is necessary to define the term *Partly Open Software Family (POSF)* in detail, which will be done in Section 2.2. This allows the derivation of specific requirements for a feasible variability modelling concept. Furthermore, there already are existing modelling concepts that are applied in the context of SPLs as well as SECOs. These concepts may serve as a basis to build a feasible concept. Moreover, it is conceivable to support building an appropriate modelling concept by combining parts of already existing ones.

Taking this as a starting point, five concrete goals can be defined by which the research question can be answered.

Comparison of SPLs and SECOs. To support a later definition of what a POSF is, it is necessary to know the exact commonalities, similarities and differences of SPLs and SECOs. These two concepts are the foundation to create POSFs as a concept in between. Therefore, a clear definition of both concepts is required. A major opportunity POSFs are supposed to offer is an improved collaboration of multiple contributing parties. With that in mind, the focus of this comparison is on the way contribution is handled and performed in SPL and SECO software families. The aim is to analyse which aspect of both software family types is desired to be part of a POSF conception, too.

Defining the concept between SPL and SECO. The intent of this goal is to define a foundational concept to work with. This concept of a POSF shall be derived from SPLs and SECOs as an intersystem type of software family – comprising the desired and avoiding the undesired (or even contradictory) characteristics of SPLs and SECOs. In consequence, specific requirements can be elaborated to find a feasible and appropriate variability modelling method. POSFs shall in some cases offer advantages over implementing an SPL or a SECO. This should be justified exemplarily.

Analyse existing variability modelling methodologies. After the clarification of what a POSF is and the derivation of specific requirements for a suitable variability modelling concept, existing techniques can be examined for promising approaches. Ideally, one existing modelling method serves as a basis for a new concept. Such a modelling method would only need some modifications to become feasible for POSF modelling. The purpose of fulfilling this goal is to be able to draft a particular mechanism for POSF modelling.

Creation of a feasible variability modelling concept. The creation of a feasible POSF variability modelling concept represents the main part of this thesis to answer the research question. Accomplishing this goal means to concretely create a suitable variability modelling concept. To prove its suitability and feasibility, a prototypical implementation of a modelling tool shall be developed first. Afterwards, this tool can be used to perform a case study, which actually proves the desired feasibility. The results need to be discussed as well.

Change impact analysis. A good model offers a useful and as-realistic-as-possible representation of the "real world", though it helps to keep the overview even of complex systems. This means making local changes to the model should have a limited impact on the model as a whole. Regarding models with multiple contributors and limited access rights for each contributor, this criterion of a good model can be ensured by avoiding interfering impacts on other contributors while editing the model. According to this scenario, a change impact analysis shall be performed for the proposed modelling concept. This analysis helps identifying ways to reduce or avoid the above mentioned undesired change impact.

1.3 Thesis Structure

This thesis is structured as follows:

Chapter 1 serves as an introduction to the topic in general and formulates the concrete research question with its derived goals this thesis should accomplish.

Chapter 2 presents the concept of POSFs by determining a viable sweet spot between both SPLs and SECOs as software family concepts.

Chapter 3 prepares for creating a feasible variability modelling concept by means of evaluating existing modelling methodologies and concepts. This analysis should be helpful to find appropriate solutions or parts of a solution, which can be reused within that new concept for variability modelling of POSFs.

Chapter 4 comprises the creation of a feasible variability modelling concept. Additionally, the development of a prototypically implemented software modelling tool is presented in this chapter.

Chapter 5 answers the question of how changes to the model that are performed by one contributor will or could affect other contributors directly or indirectly. Ideally, contributors are unable to interfere with others, such that they are entirely independent.

Chapter 6 discusses the modelling concept presented in Chapter 4 using the implemented tool and a case study. The feasibility according to the former stated POSF requirements gets special attention.

Chapter 7 concludes the thesis by summarising its content and discussing the presented results according to the defined goals. Furthermore, this chapter provides an outlook for future work.

2 Partly Open Software Families

Chapter 1 provided an overview and a motivation of this thesis' context as well as specific goals to accomplish. The aim of this chapter is to achieve the first two goals described in Section 1.2 by giving reasonable definitions of the terms this thesis and further work relies on.

To create a contextual frame for the further work, the bordering concepts are introduced and specified in the first part of this chapter. This clarification focusses on the distinction between SPLs and SECOs, which allows deriving POSF characteristics and requirements from the mentioned bordering concepts. In consequence, a concrete definition of POSFs is assembled in the later part of this chapter followed by a justification of this new tradeoff concept showing chances and opportunities this new type of software family potentially delivers. To serve as a reference for the further work of this thesis, concrete POSF modelling requirements are given, as well.

2.1 Software Families

The general field this thesis is situated in, is the field of variable software systems. The concepts of SPL and SECOs as presented in Chapter 1 share one conceptual commonality: the existence of common and differential software artefacts. In consequence, each of the shown types of variable software systems actually describes a set of concrete software products that are derivable out of this software system.

The umbrella term of these different types of variable software systems is *software families*. Each member of a family differs from each other (variability), but there still are some specific commonalities among each and every family member.

Software families combine the advantages of both monolithic standard software systems and per-customer developed individual software systems to a tradeoff to steer clear of the otherwise common drawbacks. In contrast to individual software, standard software is more affordable while potentially not exactly meeting the requirements of each customer. Individual software, on the other hand, fits the specific customer requirements while accepting a higher monetary effort. The aim is to build a software system that fits the customer's needs and requirements as effectively as possible while reducing the development effort compared to individually developed software, leading to a cost (and price) reduction. From a developer's perspective, van der Linden et al. stated the main reasons to design a software family in contrast to standard or individual software systems as follows:

> "The main arguments for introducing software product family engineering are to increase productivity, improve predictability, decrease the time to market, and increase quality (dependability)." [Lin+04, p. 110]

Even though single standard software and individual software each have their right to exist, it can be stated that variable software systems serve a promising market with a high demand for well-fitting and affordable software.

Giving a general definition of the term *software family* is a challenging task as the term itself is blurry. Such a definition requires to find a certain degree of commonalities that are necessary to call a set of software programmes a family, but a single number for that degree is not sufficient. Parnas [Par76] took up the assumption that the commonalities among a software family are more than the amount of variability. This makes software family engineering advantageous compared to the separate development of different software products, because common design decisions can be made on early development stages which leads to cost reduction at development time and during maintenance. That idea seems promising on the one hand, but can be questioned on the other considering the following example: Given a smartphone operating system such as Android, iOS or Windows Phone as the common platform of a software family and the installable apps as variability, there is no constraint which limits the maximum number of installed apps per phone. This would lead to a denial of considering a phone with installed apps to be a member of the software family in case a specific number of installed apps is reached so that the amount of apps/variability is "more software" than the operating system/commonality. That is why building a definition for software families founding on that commonality-variability-relation could be error-prone.

To introduce a general and abstract definition of software families, the following Definition 2.1 is proposed:

Definition 2.1: Software Family

A software family is a set of software programmes whose aggregate of properties comprises a common proper subset throughout all programmes of that software family. The larger the cardinality of that subset, the more related are the programmes in that family.

Instead of giving a fixed percentage or number of common software properties that defines whether a programme belongs to a software family or not, this definition introduces a gradual approach. The more common properties a set of programmes shares, the stronger is the relationship between these programmes to confidently call this set a software family. However, in practise, the exact measurement of common software properties is less important, as software families are usually seen as software derivates out of one common development lifecycle. For instance, different variants of a software are usually seen as a software family, e. g. Autodesk AutoCAD LT, AutoCAD Architecture and AutoCAD Electrical as three variants of the AutoCAD product portfolio from Autodesk[1].

For the sake of completeness, the often cited Definition 2.2 of software families by Parnas should not be left out, but is less focused on the structure of a software family than the priority of common and differentiating properties.

Definition 2.2: Software Family (acc. to Parnas)

"Program families are defined (analogously to hardware families) as a set of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members." [Par76]

2.1.1 Variability

This thesis mainly deals with variable software systems under the umbrella term of software families. Section 2.1 already presented the idea of software families and software family engi-

[1] More information regarding the AutoCAD product palette: `http://www.autodesk.com/products/all-autocad`

neering in general. What is missing yet, is a clear definition of what *variability* exactly is in terms of software. While commonalities – as used in Section 2.1 – describe all software properties that are (mandatorily) shared across one software family, variability means everything that is supposed to vary, hence, is not common to each and every software programme of one family.

Pohl et al. differentiates between two parts of variability: *variability subjects* and *variability objects*. Definitions 2.3 and 2.4 are given accordingly.

Definition 2.3: Variability Subject
"A variability subject is a variable item of the real world or a variable property of such an item." [PBL05, p. 60]

Definition 2.4: Variability Object
"A variability object is a particular instance of a variability subject." [PBL05, p. 60]

In conjunction, both of these terms define variability as it is used in this thesis. To illustrate both terms exemplarily, the content of a glass can be seen as a variability subject, while the possible content, e. g. apple juice, black tea or lemonade, is the variability object. In terms of software, a variability subject could be – for instance – a mail inbox protocol and an appropriate object could be the Post Office Protocol (POP) or the Internet Message Access Protocol (IMAP).

One concrete member of a software family can be referred to as *variant*, which means that the software programme comprises all of the commonalities defined by the family, and each variability subject is *configured* to match a variability object. Regarding the drinking glass example, binding the black tea as a variability object to the content subject means that black tea is actually in one specific glass.

2.1.2 Software Product Lines

One manifestation of software families is the concept of Software Product Lines (SPLs). The introduction in Chapter 1 characterised SPLs as a software family that allows the derivation of concrete software products by combining the common parts of the family with a selection of variable parts out of a closed set of options. This describes the fundamentals of an SPL very abstractly and in a generalised way. To elevate this into a more conceivable context, Pohl et al. [PBL05] defined *Software Product Line Engineering (SPLE)* by means of the terms *platform* and *mass customisation*, which can be seen as the essence of the purpose SPLs serve.

In the sense of software, a platform is the common structure or system, further software can run on to serve the user's needs and requirements. Pohl et al. define platforms as given in Definition 2.5 as a base technology. In the context of an SPL this forms the commonality.

Definition 2.5: Platform
"A platform is any base of technologies on which other technologies or processes are built." [PBL05, p. 6]

For a broad customer base, there often is no one-fits-all solution. Assuming there is just one solution available, this often just partly solves the customer's individual problem. To get a better fitting solution, the customer needs an individual one, but the (monetary) effort is much

higher compared to standard solutions. That is why companies in many fields offer a range of similar yet different products to keep the effort (development cost and sales price) moderate but the offered solution better-fitting. This idea can be seen as a tradeoff between mass production and customisation alias tailoring – or *mass customisation* for short. Pohl et al. used the mass customisation term originally coined by Stanley Davis in 1987 and similar to the definition by Tseng and Jiao [TJM96]. To use within this thesis, Definition 2.6 is proposed to describe the term. In the context of an SPL this enables variability.

Definition 2.6: Mass Customisation

"Mass customization aims to provide customer satisfaction with increasing variety and customization without a corresponding increase in cost and lead time." [TJM96, p. 153] Hence, mass customisation allows the production of goods and offering services that meet the customer's individual requirements with almost mass production efficiency.

Combining both ideas, platforms and mass customisation, it is possible to formulate a definition of SPLE as Pohl et al. do in Definition 2.7.

Definition 2.7: Software Product Line (Engineering)

"Software product line engineering is a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customisation." [PBL05, p. 14]

An industry that massively performs product line engineering in different abstractions is the automotive sector. Multiple car models of one manufacturer can share one single common platform with just a few modifications. From a customer's perspective, there are many different individualisation options available when buying a new car. For instance, it is usual to choose from a variety of different motorisation options, between manual or automatic gear shifting as well as a set of different infotainment options such as radio and navigation combinations.

When taking a look at the software industry, a typical example for SPLE is the enterprise software SAP Business ByDesign[2]. Business ByDesign is a customisable Enterprise Resource Planning (ERP) software solution that is offered in the form of a Software as a Service (SaaS) application. Figure 2.1 shows the online configurator, which can be used to create an individual variant of the product. The configurator provides a set of features that can either be selected or deselected. For example, the feature *Sales Quotation* magnified in Figure 2.1 adds, if selected, functionality to the product instance that allows creating and managing quotations.

Typical for the development procedure of an SPL is following the two-lifecycle process as shown in Figure 2.2. This model comprises two interrelated lifecycles: the *domain engineering lifecycle* and the *application engineering lifecycle*. During the domain engineering process, all of the required software artefacts are developed, which includes the common and all variable parts. The application engineering process, on the other hand, focusses on using the prepared artefacts to build up a concrete variant of the product family.

Both lifecyles include the usual development phases known from the waterfall model: *analysis*, *design*, *implementation* and *testing*. The key difference to the waterfall model is the intention of each phase. During the domain engineering cycle, these phases generate the later used artefacts (shown in the centre part of Figure 2.2). The domain analysis phase leads to a set of requirements, the domain design generates design artefacts, the domain implementation delivers the

[2]SAP Business ByDesign product information at `http://go.sap.com/product/erp/business-bydesign.html`

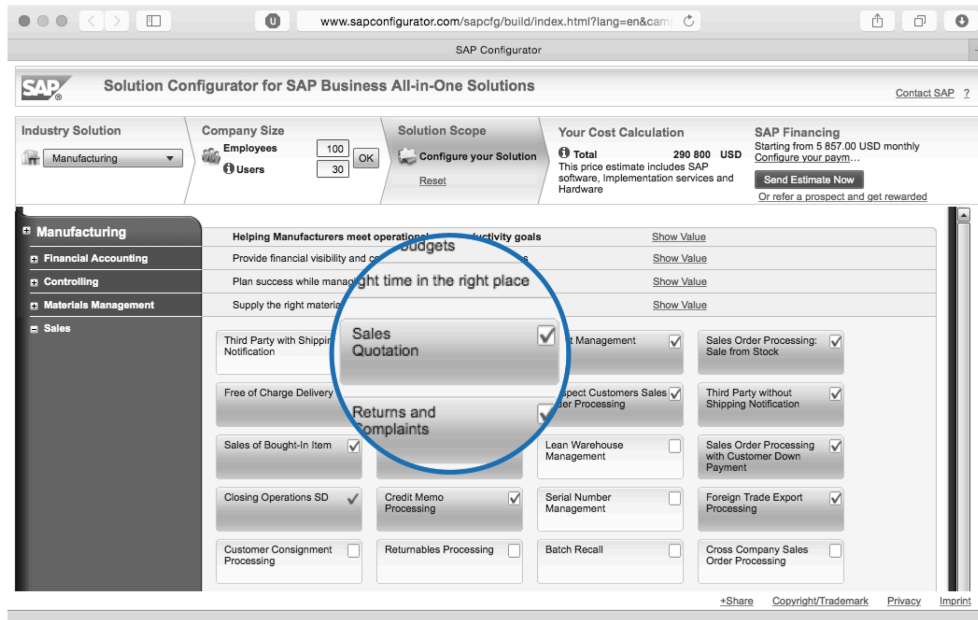


Figure 2.1 – SAP Business ByDesign configurator that allows the product individualisation, screenshot from `https://www.sapconfigurator.com/`

actual software artefacts and, finally, the domain testing phase produces reusable tests. On the other hand, the application engineering cycle's attempt is to use these artefacts for the final product variant.

One important characteristic of SPLs to mention here is that the actual contribution to an SPL can be done by providing artefacts during the domain engineering process, which is controlled by the *product management*. In consequence, there necessarily is one centralised controlling instance with the exclusive right to influence the product line. Figure 2.3 visualises that as a supplier network. The integrator in the middle represents the product management instance as mentioned above. The actual software parts are delivered by one or more suppliers (left part of the figure), which can be an internal development team or external (e. g. sub-contracted) contributors. The integrator assembles the actual product line that, finally, allows customers to get an individual variant (right side).

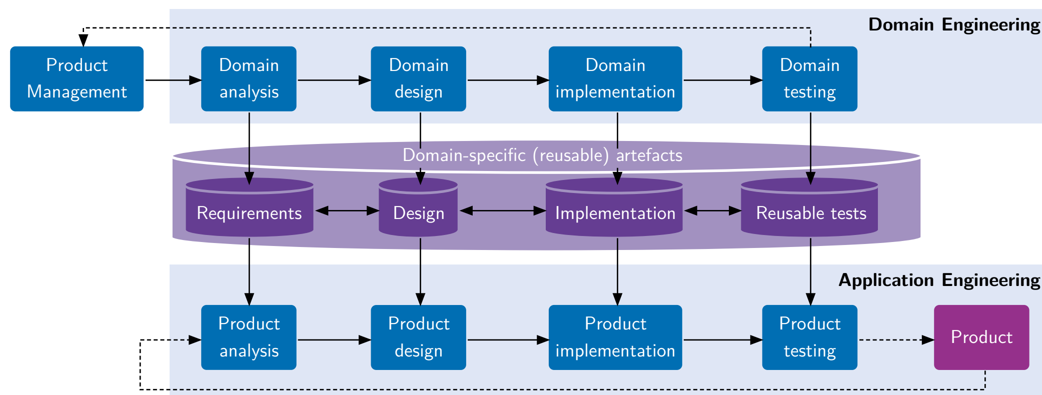


Figure 2.2 – Two-lifecycle process of SPLE, cf. [SS13, p. 25; Lin07, p. 7]

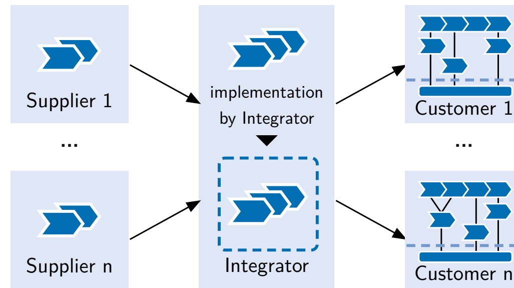


Figure 2.3 – SPL supplier network based on [Lan+08, p. 3]

In case of the former mentioned AutoCAD example, Autodesk serves as an integrator of the different features the offered products comprise. Regarding the AutoCAD suppliers: Whether the full developmental effort is done internally or if there are additional external (sub-contracted) contributors is not visible to the customer. He may only choose between different offered variants of the AutoCAD lineup of products with or without a domain focus (e. g. architecture, maps and electrical systems).

2.1.3 Software Ecosystems

Software Ecosystems (SECOs) are another concept of software families that appears as a contrasting concept to product lines. On the one hand, an SPL comprises a system to allow the derivation of multiple product variations with the aim of reducing the developmental effort compared to the individual development of each product. This allows a software vendor to offer better fitting solutions to his customers' problems.

On the other hand, another approach to create customised software solutions is to open up the development process to external contributors. The idea is to separate the distribution of product commonalities and variability of one software family. Therefore, there is one party that provides the software family's commonalities in form of a platform in the sense of Definition 2.5. This platform offers opportunities to customise the software by supplying variability subjects in the sense of Definition 2.3. The variability according to Definition 2.4 can, finally, be offered by external vendors.

Well-known examples of SECOs are the prevailing smartphone operating systems iOS from Apple and Android from Google in conjunction with their apps. Apple iOS is technologically based on the desktop operating system Apple OS X (i. e. its Darwin foundation), which is a UNIX based system. Apple uses iOS on their own smartphone and tablet computer systems. Android from Google is an open-source variant of the Linux operating system and is used by many different smartphone and tablet computer manufacturers. On both platforms, variability is realised by installing so called apps onto the user's device. An app usually implements a few functionalities that extends the capabilities of the device it is installed on.

Another popular example of a SECO in the area of software development and programming is the Eclipse Integrated Development Environment (IDE). Eclipse comprises a comprehensive framework that allows flexible extension with new functionalities. Therefore, a large community within the Eclipse ecosystem has been developed over time, which supplies a large amount of plugins.

This thesis uses the following Definition 2.8 to define the term *Software Ecosystem*.

Definition 2.8: Software Ecosystem (SECO)

A Software Ecosystem is a software family with platform and extension contributors. Platform contributors supply a common fundamental technology that is shared by each software family's member. Extension contributors deliver variable software parts that are based on the shared platform and allow the creation of distinct software family members (variants). There is no centralised software (product) management, thus, each contributor can act mostly autonomously.

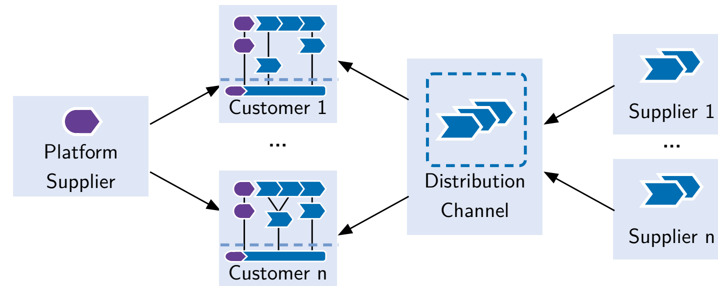


Figure 2.4 – SECO supplier network in contrast to Figure 2.3

Figure 2.4 visualises the supplier network of a SECO in contrast to Figure 2.3 for SPLs. The key difference to an SPL is the decentralised development approach that is realisable here. The common platform is delivered by one dedicated platform supplier (outer left side of the figure). He delivers the necessary software family's commonalities, which are part of every software variant, to all customers. The variable part is contributed by external suppliers (outer right side of the figure). The way how suppliers distribute their software parts is not restricted by the definition of SECOs, but may be defined by the platform supplier. For instance, in case of the Apple iOS platform, the contribution of apps is only possible through Apple's own App Store (despite some exceptions). In contrast, in case of Google's Android mobile platform, apps can either be distributed using Google's own Play Store (which is their pendant to the App Store form Apple) or using any other distribution channel such as a website. The distributed apps are contributed by external developers.

2.1.4 Commonalities and Differences of SPLs and SECOs

The basic comparison of both software family concepts, SPL and SECO, leads to the overview given in Table 2.1. The commonalities of both concepts mainly refer to the characteristics of software families in general. Thus, both concepts offer the opportunity to create variable software systems with shared commonalities (called platform). That is why both concepts belong to the category of standard software as the opposite of individual software. Standard software is more affordable than individual software as the development expenses can be spread over a large number of customers, whereas individual software has only one or a few customers, so that the per-customer costs are significantly higher. The downside of standard software compared to individual software is a deficiency in matching the customer's requirements.

Variability Management. A main difference between SPLs and SECOs is the contribution concept. Whereas all contributions to an SPL are channelled to one single management instance (called *Integrator* in Figure 2.3), contributions to SECOs can usually be managed autonomously.

Table 2.1 – Comparison of SPLs and SECOs

	Software Product Lines	Software Ecosystems
Commonalities		
	• Standard software (no individual software) • Software families • Comprise commonalities (shared platform) and variability • Realisation of individualised software products	
Differences		
Variability Management → Variant Space	Centralised (closed variant space)	Decentralised (open variant space)
Lifecycles	• One domain engineering lifecycle • An application engineering lifecycle for each derived product variant	• One platform lifecycle • An extension lifecycle for each extension • An application lifecycle for each variant
Direct Contributors	One (may be external, sub-contracted suppliers)	Many (at least one platform supplier, extension suppliers)
Business Model (Generalised)	• The vendor sells a product that can be customised to some extent.	• The platform vendor earns money by offering a platform (e.g. by selling it or by offering a commission-based extension distribution channel, i.e. App Store). • Extension vendors sell their platform extensions to customers using an appropriate distribution channel.
Constraint Model (Change Impact)	Variability constraints can be set across the entire product line. Potentially high change impact.	Tendency to avoid constraints across extensions. Potentially lower change impact.

Only contributions to the software platform are organised in a centralised way comparable to SPLs, because there is only one platform vendor per ecosystem visible to the customer. If there are multiple parties, which contribute to the software platform, they need to be closely related similar to the contributors of SPLs. Hence, there is one single platform leader, who acts similar to the integrator of an SPL. Worth to mention here: The assembling process of a specific software variant differs between SPL and SECO based software. In case of a product line, the customer gets access to a closed set of variable elements he may choose from until a final, valid variant has been created. In contrast, inside of an ecosystem, the customer gets access to the platform first and, afterwards, is able to obtain variable software parts through a distribution channel.

Variant Space. Closely related to the centralised or decentralised variability management approach is the openness of the variant space. The centralised variability management leads to a variant configuration process that is based on a fixed set of variable software parts. Hence, at the moment of variant creation, all possible variants are hypothetically known in advance. In contrast, the decentralised variability management of SECOs leads to a more dynamic variant creation process. The customer begins to create his variant based on the platform by adding variable software parts supplied by external vendors. Thus, the set of available variable software parts is not fixed but open. In consequence, it is impossible to know all possible variants of the ecosystem-based software family in advance.

To resume that argumentation, the centralised variability management of SPLs leads to a closed variant space. On the contrary, the decentralised variability management of SECOs derives an open variant space.

Lifecycle. SPLs have a centralised variability management. Hence, the whole development effort that has to be done to create a product line can be organised by the integrator/vendor within one domain engineering lifecycle. That lifecycle, therefore, comprises the development of common and variable software parts at the same time. The domain engineering lifecycle is shown in the upper part of Figure 2.2. On the other hand, this is not a model that can be applied to ecosystems because of its immanent decentralised approach. This decentralisation requires splitting the domain engineering lifecycle known from product lines into multiple per-contributor lifecycles. As a result, there is one platform lifecycle and one for each externally contributed extension.

Next to the domain-based lifecycles, there exists one lifecycle for each specific variant called application engineering lifecycle. According to an SPL, this lifecycle is shown in the lower part of Figure 2.2.

Direct Contributors. In case of a product line, there is only one direct contributor. There may be more than one indirect contributor, who is external or sub-contracted by the main software developer (integrator). Concerning SECOs, there can be many direct contributors but at least one for the platform and some that contribute the variable software parts.

Business Model. Although this is not a technological difference, the distinction of underpinned business models is worth to mention. Especially, when it comes to practicability and feasibility in real world projects, the economical perspective shows off as a key aspect.

Definition 2.9: Return on Investment (ROI)

"[Return on investment is a] profitability measure that evaluates the performance of a business by dividing net profit by net worth. Return on investment, or ROI, is the most common profitability ratio. There are several ways to determine ROI, but the most frequently used method is to divide net profit by total assets." [Ent15]

The first and foremost goal of a company that sells a software product is to maximise monetary profit and Return on Investment (ROI) (cf. Definition 2.9). In every case of standard software, this can be done by a growth in customer base. That is why a software should fit the needs and requirements of as many potential customers as possible. Compared to monolithic standard software, often referred to as Commercial Off-The-Shelf (COTS) software, variable software systems offer the opportunity to address more potential customers by means of being able to serve the requirements of each customer better. Apart from that, the actual source of getting profit out of a software family may differ.

Earning money by offering an SPL is similar to the way this is done with COTS software. Selling concrete products generates revenue. A SECO offers more flexible business models to generate revenue for each contributing party. A platform vendor may sell his platform to earn money and/or he can offer an extension distribution channel as a commission-based marketplace. Such a marketplace enables external contributors to sell their software parts. For instance, the latter is usual in the area of smartphone operating systems. Additionally, external variability

contributors may earn money directly by offering their software to end-users in contrast to the indirect model implied by Figure 2.3.

Constraint Model. Regarding the change impact, the difference in the constraint model is crucial. A high degree in interrelation of the offered software parts can lead to a high change impact that makes it difficult and expensive to maintain a software family. Particular constraint restrictions might be useful to lower the change impact. Regarding SECOs, variability constraints are tendentially contributor-internal. Hence, avoiding constraints between multiple extensions lowers the change impact significantly over the non-restrictive constraint model in SPLE. Nevertheless, there is no conceptually forced constraint restriction in ecosystems. The ecosystem of the Eclipse IDE proofs that, because there are many inter-extensional constraints. For instance, an extension for file versioning (Subclipse[3]) requires another extension as an Apache Subversion (SVN) driver.

To sum up this comparison of SPLs and SECOs, it is crucial to recognise that in both cases software products are customisable, but the process of customisation is done differently. Especially the time the customisation takes place differs significantly. Referring to Figure 2.3, in case of an SPL, the customisation is done by configuring a product variant on the vendor's side. Hence, the vendor offers the possible variability options to the customer from which he is allowed to choose from. Apart from that, there is no configuration process on the vendor's side regarding a SECO. Referring to Figure 2.4, the individualisation of the customer's product is done *after* retrieving the platform and by means of adding a selection of software fragments (referred to as extensions) obtained through a distribution channel and offered by external software suppliers.

2.2 Partly Open Software Families

The key difference between both software family concepts, SPL and SECO, is the variability behaviour and structure. While a product line is

- managed by one vendor,
- based on a centralised approach and
- created with a closed variant space,

an ecosystem is

- managed by its multiple contributors (although the platform vendor has a key role herein),
- based on a decentralised approach and
- created with an open variant space.

Both types of software families have their advantages over the other. For instance, an SPL allows a developer to keep full control over the offered product variants. There is no external software dependency. In contrast, a SECO comprises a community that supplies functionalities to a platform, which indirectly increases the usefulness or attractiveness to customers of a platform without the vendor's direct effort. Additionally, the customers' requirements can potentially be matched more accurately.

[3] Further information regarding Subclipse: `https://marketplace.eclipse.org/content/subclipse`

One underlying question of this thesis is, whether it is possible to design a kind of software family that combines the advantages of both prevailing types. This means to have a software family which is designed in a way similar to a product line and similar to an ecosystem, as well.

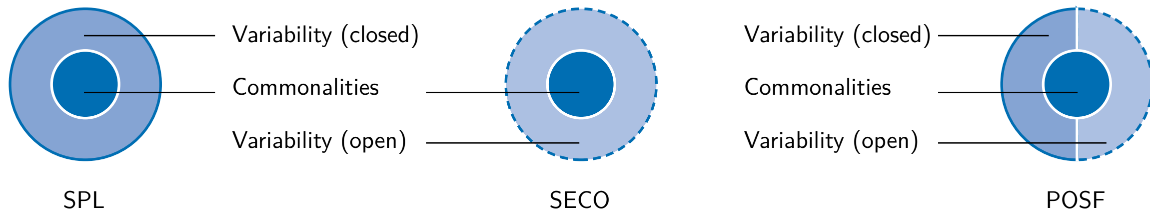


Figure 2.5 – Pie chart visualisation of SPLs, SECOs and POSFs

Designing a concept of software families that combines characteristics of SPLs and SECOs means to cope with the centralised and decentralised variability management approach at the same time. Hence, a resulting concept encompasses closed and open variability assets equally. To get an idea of this concept, Figure 2.5 serves as a highly generalised visualisation of the mentioned software family types. Each software family concept is displayed as a layered pie chart. The inner part or the core represents the common software parts that each software family includes. This is common to each software family.

The two pie charts to the left represent the two prevailing software family concepts as described in Section 2.1.2 and 2.1.3. In case of the SPL, the outer layer has a solid outline, which stands for the closed variability assets. In contrast to this, the SECO pie chart has an outer layer with a dashed outline, which shows the open variability approach.

The idea of a new kind of software family combining the closed and open variability approach is displayed as the right pie chart. The outer layer is separated into a closed and an open variability partition. That means, the software family is partly closed and partly open according to its variability assets. The relative amount of closed and open variability does not need to be balanced as the even partitioning in the figure suggests. This concept of software families will be called *Partly Open Software Family (POSF)* throughout this thesis. Definition 2.10 proposes a concrete definition of this new software family concept, which serves as the foundation for further argumentation within this thesis.

Definition 2.10: Partly Open Software Family (POSF)

A Partly Open Software Family (POSF) is a software family with a variable platform and extensions provided by multiple contributors. The variable platform comprises a common part that is shared throughout all members of the software family and a variable part that allows the derivation of multiple variants of that platform. Extension contributors deliver variable software parts that are based on the variable platform. The combination of variable platform and extensions allow the creation of distinct software family members (variants). The platform development is based on a centralised software (product) management, whereas the extension contributors can act mostly autonomously.

To identify the involved parties including their interconnection, Figure 2.6 shows the corresponding POSF supplier network. This figure is basically a combination of both the SPL supplier network given in Figure 2.3 and the SECO supplier network given in Figure 2.4.

The customer with his variant of the software family is shown in the middle and is supplied with software parts from the left and right hand side. The right part equals the extension

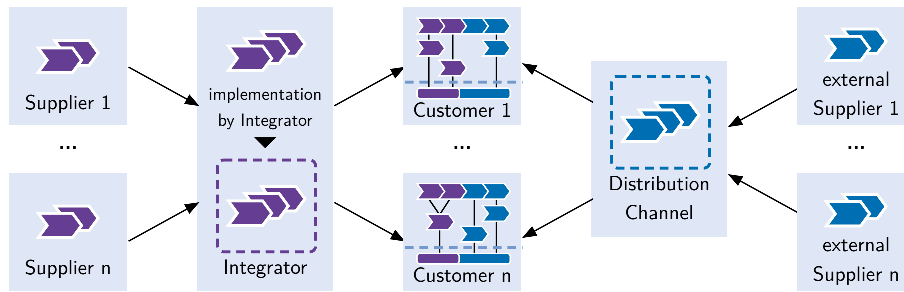


Figure 2.6 – POSF supplier network as a junction of the SPL and SECO supplier networks shown in Figure 2.3 and 2.4

distribution of the SECO supplier network. The left part replaces the platform supplier of Figure 2.4 by a complete product line similar to Figure 2.3. To distinct the involved software parts out of the closed or open pools, the (fixed/closed) SPL-based parts are purple-coloured whereas the (open) SECO-based parts are blue-coloured.

To imagine the application of that concept in practise, a combination of the former mentioned example of Autodesk AutoCAD and the possibility to install external extensions that implement supplemental functionalities could be considered. Autodesk delivers the platform in different variations (with or without domain focus) and customers are able to install extensions after purchasing an AutoCAD variant (the platform).

Table 2.2 – Characteristics of POSFs as extension to Table 2.1

	Partly Open Software Families
Commonalities (of all software families)	
	• Standard software (no individual software) • Software families • Comprises commonalities (shared platform) and variability • Realisation of individualised software products
Differentiation Criteria	
Variability Management → Variant Space	Partly centralised, partly decentralised (open variant space, although closed platform variant space)
Lifecycles	• One platform domain engineering lifecycle • An extension lifecycle for each extension • An application engineering lifecycle for each derived variant
Direct Contributors	Many (one platform supplier with optional external or sub-contracted platform suppliers, multiple extension suppliers)
Business Model (Generalised)	• The platform vendor sells a variable platform that can be customised to a certain extent. • Additionally, the platform vendor might offer a commission-based extension distribution channel. • Extension vendors sell their platform extensions to customers using an appropriate distribution channel
Constraint Model (Change Impact)	Variability constraints can be set across the entire variable platform. Potentially high change impact in that part of the software family. Regarding the extensions, there is the tendency to avoid constraints across extensions to lower the potential change impact.

In accordance to Table 2.1, the characteristics of POSFs have to be described here. Table 2.2 serves as a direct extension to this former table. It comprises the same commonalities part and the distinctive criteria as a combination of the product line and ecosystem characteristics.

One brief example for a potentially interesting business model should be presented here to point out the strengths of Partly Open Software Families. This shall be a car vendor that offers an infotainment system within their cars, which is developed as a POSF. Accordingly, the customer chooses different forms and functions *in advance* to placing the car order. Furthermore, the car vendor offers an extension mechanism to add certain features to the infotainment system on site, meaning after car delivery. This extension mechanism relies on a software distribution channel he set up similar to an App Store known from the smartphone domain. This distribution channel allows external developers to implement and offer car infotainment extensions to the customer. There are three potential points in that structure, which offer an opportunity to the car vendor to generate direct revenue. At first, selling a car generates profit. In addition to that, offering external developers to distribute their software may be a potential source of profit, too, and finally, each fulfilled trade via that distribution channel may be used to generate revenue. Regarding the external extension developer, he is able to sell his own developed software. In addition to that, the high degree of customisability leads to potentially attractive cars, which can lead to a broader customer base.

2.2.1 Requirements for the POSF Modelling Concept

Referring to the research question given in Section 1.2, a variability modelling technique for POSFs is what this thesis is looking for. Whereas variability management of SECOs does not necessarily require advanced modelling mechanisms because of the decentralised development approach, SPLs do have a centralised product management and – hence – require an appropriate modelling technique.

The variable platform can be seen as one product line, which requires the application of a modelling method as part of the variability management. As a POSF is not entirely closed but partly open similar to an ecosystem, the applied modelling method has to support that openness in a satisfactory way.

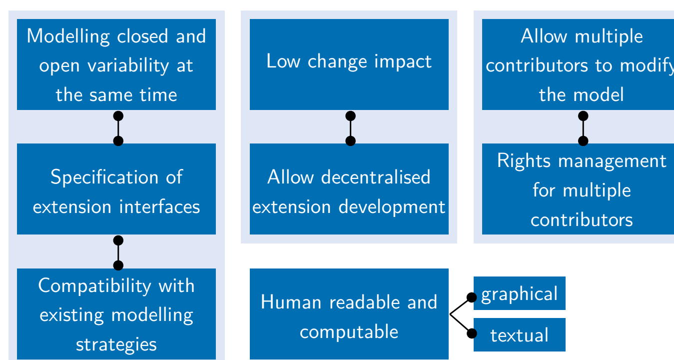


Figure 2.7 – Requirements for a modelling strategy for POSFs

Finding a feasible modelling mechanism involves defining specific *requirements* for that mechanism. Figure 2.7 lists the requirements derived from the software family concept shown in Section 2.2. They will be described in detail in the following.

Modelling closed and open variability at the same time. Because a POSF comprises both closed and open variability simultaneously, it is not sufficient to model either the closed or the open part. Thus, the model has to encompass both variability modelling approaches.

Specification of extension interfaces. Interconnected with the first requirement, there is the need for an extension specification. Regarding SECOs, the connection between platform and extensions is defined as an interface. For instance, a mobile operating system offers several Software Development Kits (SDKs) that form a framework of functionalities usable by the extensions or apps. An interface defines, in that case, what an extension is permitted and able to do. More generally speaking, an interface forms the platform-sided expectations for the extensions. Such an expectation should be formulated and included into a POSF model.

Compatibility with existing modelling strategies. Keeping a solution compatible with existing strategies and methodologies is advantageous for its feasibility in real projects. As there already are modelling methods for software families, it makes sense to derive the new POSF modelling concept from an existing kind of model.

When interpreting these first requirements in total, a process can be derived that leads to an appropriate modelling solution. As a consequence, a modification of a prevailing modelling technique is required, which supports the specification of extension interfaces to encompass closed and open variability modelling.

Low change impact. Within working scenarios with more than one contributor a good coordination is required to avoid unwanted disruptions and interference. Working on a shared model should be possible with a low level of change impact. Hence, if one contributor performs and arbitrary change to the model, another contributor, working simultaneously on the model, ideally should not be affected.

Allow decentralised extension development. While a product line has a centralised management, ecosystems do not. Furthermore, an extension developer within an ecosystem mostly works autonomously, which explicitly requires to have no centralised software management. For designing a POSF, this means that it is crucial to allow the decentralised extension development, as well, to keep those contributors autonomous. This requirement is interconnected with the demand for a low change impact. In case of a high change impact, meaning that changes of one contributor directly affect the work of another, some kind of moderation is required which leads, in consequence, to a centralised managing approach.

Allow multiple contributors to modify the model. As implied by the former two requirements, it should be possible for more than one party to contribute to the software family's model. This can be derived from the ecosystem approach as it is crucial for an ecosystem to have multiple (independently working) extension contributors.

Rights management for multiple contributors. The demand for a low change impact requires contribution rules. Hence, there needs to be some kind of rights management that permits/denies changes to the model depending on the contributor's role. This can be done in accordance to the supplier network shown in Figure 2.6.

Human readable and computable. The resulting modelling concept should support a computable (e. g. textual) as well as human readable (e. g. graphical) notation.

Modelling Challenges

The named requirements for the POSF modelling derive some explicit challenges, which have to be tackled within this thesis. They should be emphasised here.

The first challenge is to design a model that includes concrete variability, which is the closed part as well as interfaces to allow openness. Usually, open world scenarios are designed using ontologies, whereas closed world scenarios can be designed explicitly. The combination of an explicit design and an ontology does not seem to lead to a feasible solution as both approaches follow opposite principles. Hence, a modelling notation is required that encompasses closeness and openness in a satisfactory way.

The next challenge is the support for multiple contributors working on one model simultaneously. Not every contributor should be permitted to modify the entire model. There are restrictions that have to be applied somehow. Furthermore, the contributors should not interfere with each other when working on the model (requirement for a low change impact).

Finally, allowing extension developers to work autonomously is a challenge to cope with. Thus, their changes to the model need to be independent from the rest of the model.

2.2.2 Chances and Opportunities of POSFs over SPLs and SECOs

The application of a software family concept other than SPL or SECO only makes sense if there are specific advantages compared to the both mentioned. According to the POSF concept, the following chances and opportunities can be identified.

- Increased flexibility in developing a software family as there is no need to decide between a closed and open variability management, because both can be implemented in parallel.
- POSFs offer economic opportunities as it is possible to apply a conjunction of the usual business models known from the SPL or SECO concepts.
- Managing suppliers of a software product can be done more dynamically. Whereas the integration of a new platform supplier can be an effortful task, adding an external supplier is straightforward.
- The variability management can be highly dynamic as it can be selectively closed or opened.
- A POSF may allow to open an existing SPL without performing the switch to a SECO that might not be intended.

One field in which such a concept seems to be a promising approach is the area of in-car software. Today, it is usual that a customer is able to configure the functionality offered by the finally ordered car in the sense of an SPL. As cars offer a constantly and rapidly increasing amount of functionality out of the "digital world" – such as e-mailing, messaging and surfing the web – the introduction of apps as known from smartphones and tablets could open new horizons. Car vendors such as BMW or Mercedes-Benz have taken first steps in that direction, but their concepts are based on a limited variety of apps (BMW) or third-party-dependent implementations (Mercedes-Benz and others implement Apple Car Play or Android Auto). Moving to a

partly-open system by implementing principles known from SECOs into traditional SPLs could help to cope with the differently timed lifecycles of cars and electronic gadgets.

At the moment, the existing solutions for such in-car infotainment solutions are rather inflexible. Regarding the closed approach similar to the BMW apps, there is no possibility for the customer to extend the set of offered functionalities after buying the car. As mentioned above, the SPL configuration process is done on the vendor's side only. On the other hand, in case of the iOS or Android car integration, there is no way for a car manufacturer to influence the offered functionality. The car manufacturer only has the choice to support this smartphone integration in his cars or not. This is an insufficient solution as, for instance, the car manufacturer Porsche decided not to support Google's Android Auto solution due to data privacy issues recently [DPA15]. Nevertheless, these smartphone integration systems offer the opportunity to customers to individualise the functionalities offered by the car infotainment system. The solution could be a mixture of both mentioned concepts, above described as POSFs. POSFs allow customisation on the vendor's side as well as afterwards on the customer's side while keeping the control of the offered customisation possibilities to the vendor or developer.

2.3 Summary

This chapter served as a background introduction in the first part and presented the idea of POSFs next to SPLs and SECOs as a software family concept in the second part.

The key aspect while elaborating the POSF approach is the mixture of a closed and an open variant space as known from the prevailing software family concepts. The closed variant space results from the centralised variability management applied when engineering product lines. In contrast, the open variant space results from the decentralised variability management as known from ecosystems.

Furthermore, this chapter derived eight specific requirements for a feasible POSF modelling concept. These requirements are the foundation for the next two chapters. The following Chapter 3 will analyse existing modelling concepts to determine in how far they could serve as a basis for a new POSF modelling concept. The actual new concept will be elaborated in Chapter 4.

3 Analysis of Existing Variability Modelling Methodologies

The previous Chapter 2 introduced the concept of Partly Open Software Families (POSFs). The aim of this thesis is the creation of a feasible modelling technique for POSF variability. Therefore, it is useful to evaluate existing modelling techniques for structuring and formalising variability. Ideally, one of these techniques can serve as a basis to create a feasible concept for POSFs.

The first part of this chapter introduces a selection of widely used variability modelling methods. The second part evaluates the presented models regarding their feasibility to serve as a foundation for a POSF-compatible modelling technique. One of the given models is chosen for the later concept elaboration in Chapter 4. Therefore, the necessary model modifications are worked out at the end of this chapter.

3.1 Variability Modelling Methodologies

Modelling variability is not bound to a specific modelling method or visual representation. Thus, there are various modelling strategies that can be applied for variability modelling. Nevertheless, there are some methodologies that are widely used and very common to the product line community.

To develop a feasible modelling concept for Partly Open Software Families, it is useful to take a look at existing and commonly used models that could serve as a foundation for the newly intended model.

The following subsections will present a selection of four different variability modelling strategies that follow distinct approaches:

- Feature models
- Decision models
- Orthogonal variability models
- Common variability language

To keep the presentation of each modelling method comparable, a common scheme is applied here. Figure 3.1 shows the presented aspects.

1. *Relevance* – Why is it relevant to take a look at this specific modelling method?
2. *Notation principles* – What are the notation elements used for modelling?
3. *Basic modelling process* – How does modelling by means of this modelling method work?
4. *Application scenarios* – When is it useful to apply this modelling method?

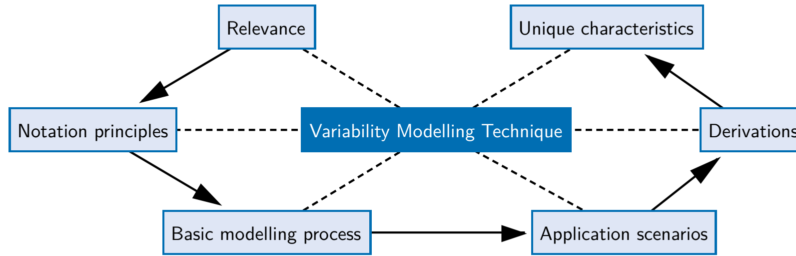


Figure 3.1 – Model presentation aspects

5. *Derivations* – Do relevant/widely used/important derivations of this model exist? If yes, they will be presented briefly.
6. *Unique characteristics* – What differentiates this modelling technique from the other models mentioned? (The unique characteristics of each presented model are presented at once after the presentation of the aspects above.)

3.1.1 Feature Models

Relevance. Feature models are widely used in the area of SPLE as they are able to represent the hierarchical structure of a product line, including all commonalities and variabilities at the same time. As software architectures are built hierarchically, feature models offer a way to carry over that architecture into the variability modelling process. Furthermore, modelling tools exist to apply feature models in real world scenarios. One of those tools is the FeatureIDE, offered as an Eclipse plugin by the Magdeburg University[1].

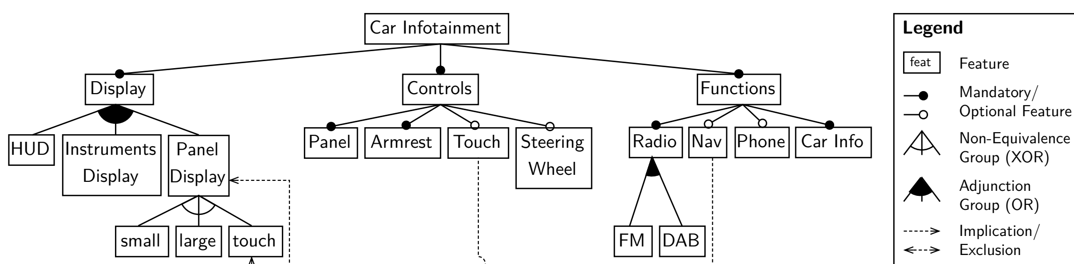


Figure 3.2 – Exemplary feature model for a car infotainment system; Kang et al. notation elements for feature models introduced within the FODA Report [Kan+90] and [CE00, p. 87 ff.]

Notation principles. Figure 3.2 shows the basic notation elements introduced by Kang et al., basically in their FODA Report from 1990 [Kan+90] as well as Czarnecki and Eisenecker [CE00, p. 87 ff.].

[1] Further information regarding the FeatureIDE can be found at `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/`

The figure shows a feature model representing a car infotainment system. This system comprises a display unit, some input control elements and certain functionalities. As a display unit, the customer should have the choice between a panel display on top of the middle console of the car, an instruments display located behind the steering wheel and a Head-Up Display (HUD). Regarding the panel display, the customer may choose between a small, large or even touch-supported version. The infotainment system can be controlled using some buttons in the middle panel area and at the armrest. Optionally, the customer can choose to have touch input as well and some controlling elements directly at the steering wheel. From a functional perspective, the system comprises radio (analogous Frequency Modulation (FM) and Digital Audio Broadcasting (DAB) radio can be chosen) and car information functionalities. If desired by the customer, the system may offer a navigation system and phone capabilities, too.

The diagram in the figure models the described system. Hence, it shows one root feature called *Car Infotainment* on top, which contains three mandatory child features (*Display*, *Controls* and *Functions*). This means, all three sub-features are required to be selected during the variant configuration process. The *Controls* top-level feature contains optional features (*Touch* and *Steering Wheel*) next to two mandatory features. This means, as soon as the *Controls* feature is selected, the two mandatory features need to be selected, too, whereas the optional features are *allowed* to be selected.

The features *Display*, *Panel Display* and *Radio* contain group relations. In case of an adjunction group (*Display* and *Radio*) the selection of *at least* one feature of that group is required. In case of a non-equivalence group (*Panel Display*), the selection of *at most* one feature is required.

To demonstrate cross-tree constraints, the model contains two implications.

$$Touch \rightarrow touch$$
$$Nav \rightarrow PanelDisplay$$

Those constraints contain format logic expressions, a concrete software configuration has to satisfy. Regarding the example, whenever the left part of each implication is true, the right has to be true as well. Hence, if the *Nav* feature is selected, the *Panel Display* has to be selected, too. An exclusion would indicate that the expression on both sides of that exclusion are required not to be true at the same time.

This car infotainment example serves as the foundation to further examples given in this chapter.

Basic modelling process. The process of building a feature model comprises

1. the creation of a hierarchical structure of all existing software artefacts (called *features*) of one product line,
2. the introduction of parent-child-relations within the feature tree and
3. the definition of cross-tree constraints.

The order of these steps is neither fixed, nor necessarily sequential but shows one possible workflow to create a model that is based on the feature modelling meta model as shown in Figure 3.3. The shown meta model consists of the basic feature modelling meta model plus two extensions (allowing attributes and cardinalities) that are discussed in the later derivations section.

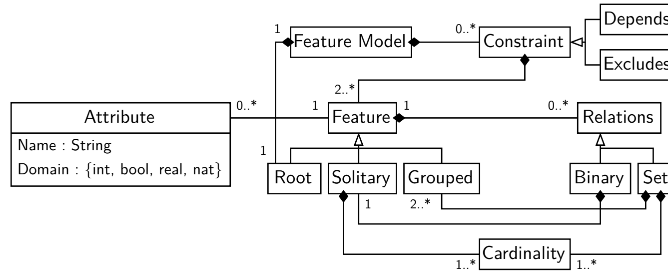


Figure 3.3 – Feature modelling meta model, cf. [RBR09]

The basic feature model comprises one root feature as a starting point. Each feature may aggregate some relations to child-features. If there are child-features, there might be one or multiple binary-related child-features or there is a group of features. The feature relations express the constraints that need to be fulfilled in order to derive a concrete software product from one SPL. Hence, they describe configuration rules – whether a feature has to be selected or not – in one product line. The selection rules apply as presented above.

Application scenarios. Feature models can be applied to model all artefacts of a product line in a tree-structured way. This includes constraints between artefacts beyond the tree-structure inherited through the parent-child-relation. These constraints are called cross-tree constraints.

One advantage over other variability modelling techniques is the easily readable structure, which can be used to express the customer's options to support the configuration process using – for instance – a top-to-bottom approach starting with the root feature.

Derivations. Feature models are widely discussed in the area of Product Line Engineering (PLE). This is why there are numerous extensions available, which support the representation of further information inside a feature model or enable increased flexibility during the modelling process.

One of those extensions is the substitution of the OR and XOR groups by using groups with annotated cardinalities. This substitution has already been respected by the meta model of Figure 3.3. Each group gets its minimum and maximum amount of sub-features that a customer has to select. To substitute an XOR group, the cardinality *[1..1]* is used so that exactly one sub-feature has to be selected. An OR group can be replaced by the cardinality *[1..n]*, whereas n is the number of available sub-features, so that at least one sub-feature has to be selected.

The car infotainment example of Figure 3.2 can be replaced by the cardinality-based feature

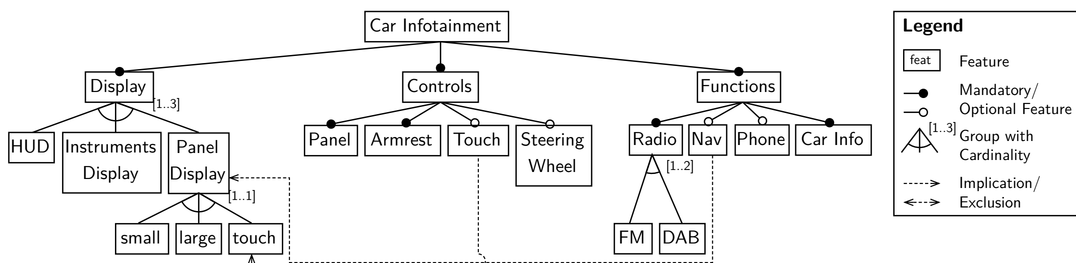


Figure 3.4 – Exemplary cardinality-based feature model showing the car infotainment system introduced in Figure 3.2

model shown in Figure 3.4. Each OR and XOR group has been substituted by its cardinality-based pendant. Additionally, cardinality-based feature models allow more flexibility during modelling as it is possible to formulate more specific constraints. For instance, it would be possible to restrict the number of selectable displays to at least/at most or exactly two out of the three available options (*[2..3]*, *[1..2]*, *[2..2]* respectively).

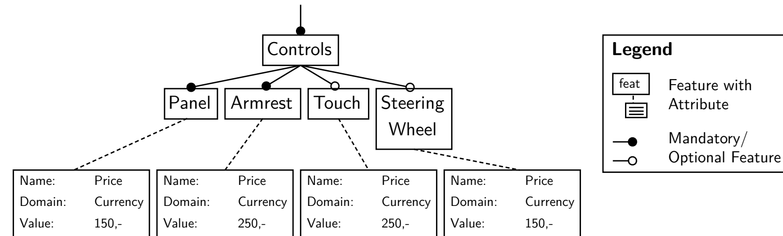


Figure 3.5 – Exemplary attributed feature model fraction showing a part of the car infotainment system introduced in Figure 3.2

Another extension that is already respected by the meta model of Figure 3.3 is the support of attributes. Attributes allow the annotation of additional feature-related information directly into the feature model. Each attribute has a *name*, a *domain* and a concrete *value*. These attributes may be used to annotate arbitrary data such as pricing information, efficiency data or code specifications. Figure 3.5 shows a fraction of the car infotainment feature model of Figure 3.2 with pricing annotations. In this case, the *Armrest* and *Touch* control input elements cost 250,00 each; the *Panel* and *Steering Wheel* control elements cost 150,00 each. The concrete currency is not specified here.

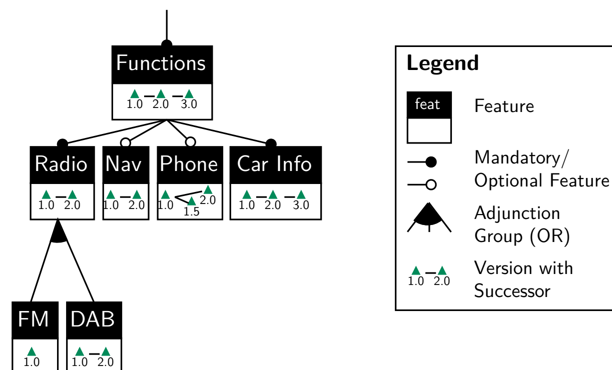


Figure 3.6 – Exemplary hyper feature model fraction showing a part of the car infotainment system introduced in Figure 3.2 based on the notation introduced by Seidl et al. [SSA14]

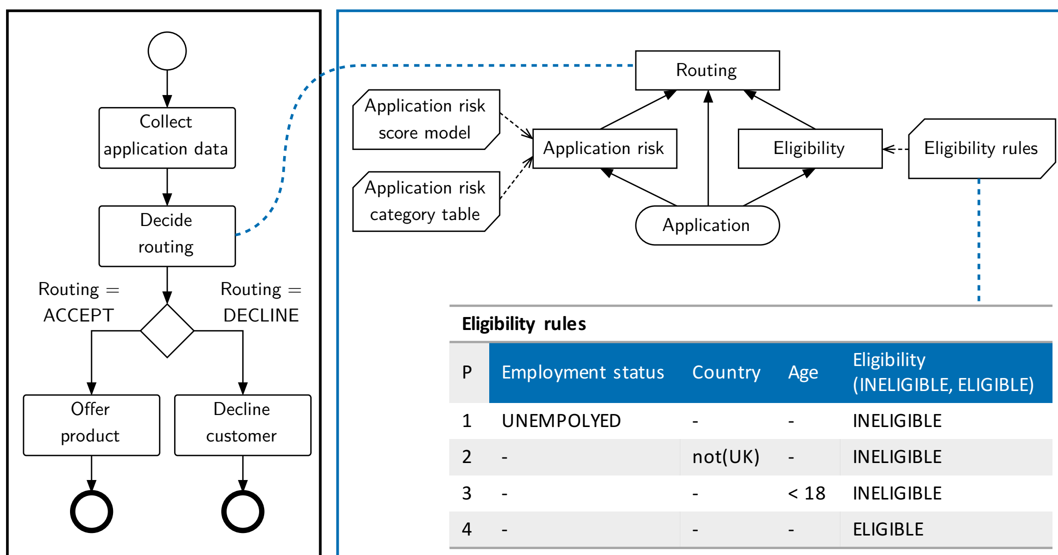
One extension to feature models that introduces a new perspective onto the underpinning product line is the introduction of *Hyper Feature Models (HFMs)* by Seidl et al. [SSA14]. An HFM supports capturing evolutionary aspects of SPLs, as there may be multiple versions of one feature available, which have been created over time. Those evolutionary processes cannot be represented by one single ordinary feature model. The extension allows adding a version history for selectable versions to each feature in the model. Figure 3.6 shows a fraction of the car infotainment example of Figure 3.2 as a HFM with versions. This would allow a customer to choose between different available and compatible versions of each feature. For instance, the customer could choose between version 1.0 and 2.0 when selecting the navigation system (*Nav*)

of desire. An appropriate application scenario for that could be a car fleet manager that wants all his cars equipped with the same navigation system, so that the actual users of the cars feel comfortable in each of the cars. Therefore, the fleet manager will order cars with *Nav* in version 1.0, regardless of the fact that a newer version became available in the meantime.

3.1.2 Decision Models

Relevance. When it comes to integrate variability into product lines, building a concrete variant of a product line means to perform multiple decisions. In conjunction with this, *decision models* offer a modelling technique to perform these decisions.

Similar to a feature model, decision models represent the valid variant space. A valid product of a product line has to obey the rules given by the underlying model. Decision models show consequences (referred to as decisions) of a set of possible given situations. In contrast to the other presented modelling techniques, decision models focus on the decision making *process* rather than the variability structure.



P	Employment status	Country	Age	Eligibility (INELIGIBLE, ELIGIBLE)
1	UNEMPOLYED	-	-	INELIGIBLE
2	-	not(UK)	-	INELIGIBLE
3	-	-	< 18	INELIGIBLE
4	-	-	-	ELIGIBLE

Figure 3.7 – Exemplary DMN model showing the basic notation elements and the relation to BPMN, cf. [Obj14]

Notation principles. To visualise the modelling process, Figure 3.7 shows the basic decision model elements within a brief example. To present the basic elements in short, the example is based on the exemplary description of the Decision Model and Notation (DMN) by the Object Management Group Inc. (OMG)[Obj14, p. 22]. The car-infotainment-related example is given afterwards.

The left part of Figure 3.7 shows an optionally underlying Business Process Modelling Notation (BPMN). This BPMN models a simple workflow that decides whether to offer products to a customer or not. This workflow requires a decision process that decides, whether a customer should be accepted or declined by the system.

The right part of the figure models this decision making process. The root decision here is *Routing*, which is separated into the two sub-decisions *Application risk* and *Eligibility*. The input for each decision element of the model is an *Application*. The "intelligence" of the decision

model is offered by decision logic that controls the decisions. In the given example, the logic elements *Application risk score model*, *Application risk category table* and *Eligibility rules* control the decision making process.

The logic input element *Eligibility rules* is presented in detail. This logic element comprises a decision table as shown in the lower part of the figure. This logic element presents different situations or cases (form 1 to 4) with their characteristics regarding *employment status*, *country* and *age*. For each case, the decision table leads to a consequence (referred to as decision) as shown in the last column *eligibility*. The example shows three cases that lead to a customer's ineligibility. The customer is ineligible if he is unemployed, does not live in the United Kingdom or is under the age of 18. In every other case, the customer is considered eligible.

Table 3.1 – Example of a decision model based on the car infotainment example

Decision Name	Description	Type	Range	Cardinality	Rule
Display	Which display shall be used?	Enum	HUD | Instruments | Panel	1:3	**IF** Display.contains("Panel") **THEN** PanelDisplay.isEmpty() == false;
Panel Display	Which panel display?	Enum	small | large | touch	1:1	
Touch	Touch controls desired?	Bool	true | false		**IF** Touch.getValue == true **THEN {** Display.addValue("Panel"); Panel.setValue("touch"; **}**
Steering Wheel	Steering wheel controls desired?	Bool	true | false		
Radio	Which radio band shall be supported?	Enum	FM | DAB	1:2	
Nav	Navigation system inside?	Bool	true | false		**IF** Nav.getValue == true **THEN** Display.addValue("Panel");
Phone	Phone support desired?	Bool	true | false		

An example based on the prior introduced car infotainment system shall be given, too. Table 3.1 represents the decision options and rules equally to the feature model of Figure 3.2. This example does not comprise a structured decision process, but models all variability assets of the proposed infotainment system. Selection relations of the feature model are mapped to *rules* (right column).

To carry over decision models into the field of software variability, the given input (in the example of Figure 3.7 the employment status, country and age) would represent a customer's choice over a variable set of options (e. g. software features) and the decision making process would result in a concrete decision of how to assemble the specific software variant that fits the customer's choices.

Basic modelling process. The modelling process can be done complementary to a business process model to formally define decision processes. To model variability in software families, an underpinned BPMN is not required. Every variability point of a software family can be modelled as a DMN model.

The modelling process itself comprises building a tree-structured decision process. Hence, there is one root decision to be made relying on sub-decisions. Each decision has input elements

that lead to the actual decision. Decision logic is the main part of the decision making process and can be specified as a decision table.

Application scenarios. Decision models are defined as DMN by the OMG[Obj14]. The DMN specification has been developed to seamlessly work together with the BPMN, which is an OMG standard as well. BPMNs are models to visualise business workflows in conjunction with business rules. Those workflows can be part of a software system as well. Some parts of those workflows require a systematic decision making, which can be modelled using a decision model as discussed here.

In terms of variability modelling for software families, decision models help to specify the available options and constraints to decide for or against the specified options.

3.1.3 Orthogonal Variability Models

Relevance. The Orthogonal Variability Model (OVM) provides the structure of variation points of a software and constraints between multiple variation points. These models ignore commonalities of a software application but focus exclusively on the variability. Modelling each variation point solely offers the opportunity to model orthogonal variability such as the choice between a messaging service (e. g. Short Message Service (SMS) and Multimedia Messaging Service (MMS)) or the choice between phone operating systems (e. g. Google Android and Apple iOS). Feature models that contain these aspects inside of a monolithic model would grow exponentially when inserting an additional variability dimension.

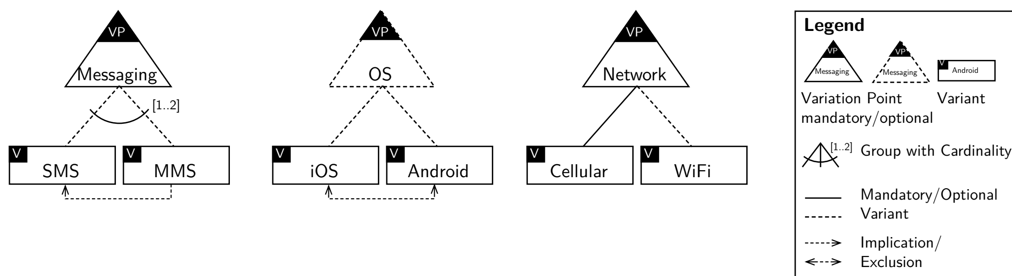


Figure 3.8 – OVM notation elements based on [RBR09, p. 6]

Notation principles. To create a model based on the proposition of Pohl et al. [PBL05; PM06], Figure 3.8 shows the available notation elements. Each variation point is represented by one flat (two-layered) tree-structure. The top element of this structure specifies the variation point of interest. This element is displayed as a triangle with a *VP* marker and the name of the variation point. Variation points that are mandatory for a software application have a solid border, whereas optional variation points have a dashed border.

The available options for each variation point, the variants, are listed as sub-elements of a rectangular shape in each *VP*-tree. A variant is marked with a upper-left *V* and contains the variant name.

The connections between the root node and the available sub-nodes in each tree specify the selection constraints for each variation point. Solid connection lines visualise mandatory selection elements, dashed connections represent optionally selectable variants. If there are restrictions regarding the number of selectable variants, this is marked using annotated cardinality that

defines the restriction. Implications and exclusions are modelled similar to those in feature models.

The exemplary Figure 3.8 shows all of these elements exemplarily.

The left tree shows the mandatory variation point *Messaging* with its variants *SMS* and *MMS*. The angle signalises that at least one of both options needs to be selected, but it is possible to choose both options as well. The implication from *MMS* to *SMS* forms a dependency between both variants. Hence, if *MMS* is chosen, *SMS* has to be selected, too.

The tree in the middle shows the optional variation point *OS* with its optional variants *iOS* and *Android*. There is no explicit selection restriction given but the exclusion between both offered variants indicate that *iOS* and *Android* cannot be chosen at the same time.

The right tree shows the mandatory variation point *Network* with its variants *Cellular* and *WiFi*. As there is a mandatory connection to *Cellular*, this variant has to be part of the software application. The *WiFi* variant can be chosen additionally.

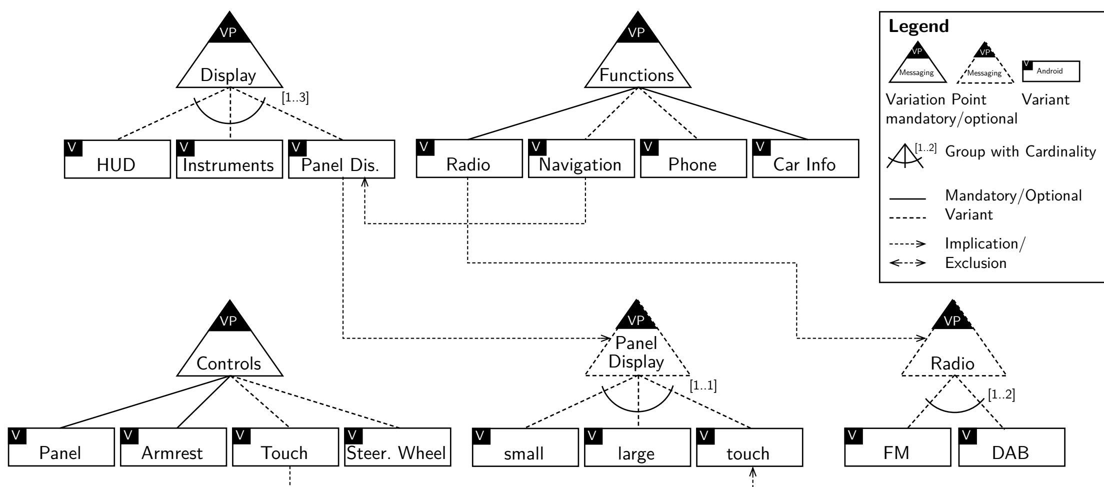


Figure 3.9 – Exemplary OVM model representing the car infotainment system introduced in Figure 3.2

Figure 3.9 shows an OVM representation of the car infotainment example of Figure 3.2. As the features *Display*, *Controls* and *Functions* are mandatory, these three features are mandatory variation points of the shown OVM.

The sub-trees of the original feature model at *Panel Display* and *Radio* are represented in separate, optional variation points inside the OVM. These optional variation points are connected to the other variation points using implications. The two implications of the original feature model are carried over analogously onto the OVM.

Basic modelling process. The modelling process comprises the following steps:

1. List all variation points of a software application.
2. Decide, which variation points are mandatory for the software application. The other variation points are optional but may be required by specific choices of mandatory variation points.
3. Enumerate all possible variants for each variation point.

4. Define the selection constraints for each group of variants.
5. Define the constraints between multiple variants and variation points (implications, exclusions).

Application scenarios. OVMs can be used to model every variation point of a software family with its possible variants and the constraints between multiple variants and variation points.

If the software structure should not be modelled within one single tree-structure, an OVM could be an effective modelling strategy. Modelling orthogonal variability subjects would lead to an exponential blow-up in case of creating a feature model. In those scenarios, OVMs are advantageous.

3.1.4 Common Variability Language

Relevance. The Common Variability Language (CVL) is a modelling strategy proposed by Haugen et al. [Hau+08] to introduce a universally applicable standard for variability modelling within any desired realisation model. This versatile approach shall help to support modelling variability as part of the realisation process of an SPL.

Notation principles. A Variability Specification (VSpec) tree comprises the same elements as a feature model does: Singular features (tree leafs) are called *options*, whereas features with child-features are called *groups*. Cardinality information and mandatory as well as optional binary relations allow to define selection constraints in that variability model. Cross-tree constraints are visualised as logic expressions connected to the root element of the model.

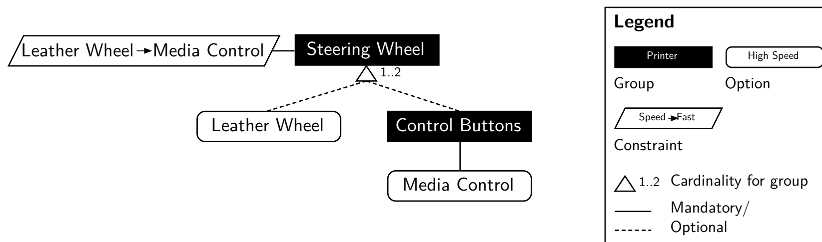


Figure 3.10 – CVL notation elements, cf. [HW12, p. 8]

Figure 3.10 shows the mentioned elements as their visual representation. Groups, options and constraints are distinguished by their shape. Groups are displayed as rectangles, whereas options are visualised as ellipses or rectangles with round corners. To differentiate both elements better, Figure 3.10 shows groups as black boxes. Constraints are displayed as parallelograms with their logical expression inside.

Mandatory and optional selection constraints are displayed similar to those used in OVMs. Mandatory relations are displayed as a solid line, whereas optional relations are displayed as dashed lines.

Groups and their cardinalities are visualised as a small triangle below the group shape with its cardinality information aside.

To give a brief example of how a complete VSpec tree may look, the car infotainment example of Figure 3.2 can be transferred to a CVL-compliant VSpec tree. Figure 3.11 shows that model. The major difference between these two models is the way cross-tree constraints are displayed inside the model.

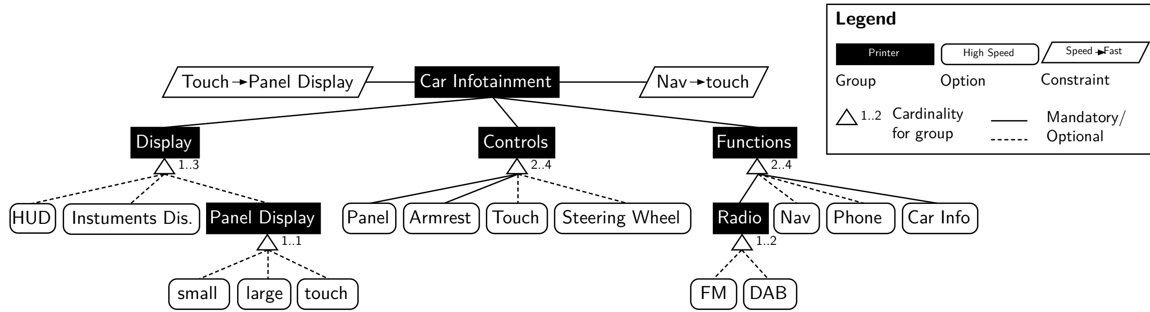


Figure 3.11 – CVL example showing the car infotainment system introduced in Figure 3.2

Basic modelling process. The variability modelling process for CVL models partly relies on the software modelling strategy, as VSpecs are formulated during this modelling process.

The VSpec formulation is similar to the definition of an variation point within OVMs as illustrated in Section 3.1.3. Each variation point is modelled as one single VSpec. The corresponding constraints are part of this specification as well.

A complete variability model can be compiled by assembling all defined VSpecs to a VSpec tree that may be structured very similar to feature models.

Application scenarios. The CVL is intended to be used to model variability directly in conjunction with realisation models, which are generated during the design phase of the software family development project. Variability information can be added as an annotated *VSpec* in form of an overlay into an existing model such as an Unified Modelling Language (UML) class diagram or similar.

3.1.5 Unique Characteristics of the Presented Modelling Methods

Unique feature modelling characteristics. The approach of representing a whole product line with its common and variable parts is characteristic for feature models compared to other variability modelling techniques. This representation is done using one tree-structured hierarchy which offers the opportunity to model the tree similar to the actual software to built.

Unique DMN characteristics. The DMN focusses on the process of choosing between options that are offered by means of software variability. The software structure does not play a significant role in those models. This clear focus on the process instead of the structure distinguishes DMNs from the other models presented in this chapter.

Unique OVM characteristics. The unique characteristic of the OVM modelling technique is the pure focus on variability modelling. Each variation point is modelled in a flat tree-structured way. The interconnection of the *VP*-trees is loose and explicitly non-hierarchical.

Unique CVL characteristics. CVL models follow a composite approach that makes it easy to integrate variability information inside of other software design models as an overlay such as UML models. The actual variability model is a composition of all variability information overlays that were integrated into other realisation models. The composite approach is comparable to the one shown with OVMs, but the intention differs. While OVMs follow a loose composition of

variation points to avoid a model blow-up when modelling orthogonal software properties, the CVL is designed compositionally to support the direct underpinning of variability information to the according elements inside the realisation model. Hence, the interconnection of multiple variation points within OVMs is done directly using implication and exclusion arrows. On the other hand, constraints are annotated in a decentralised logical expression on each VSpec.

3.1.6 Brief Summary of the Presented Modelling Methods

Table 3.2 – Brief summary of the presented modelling methodologies

Model	Brief Description
Feature Model	• Comprises commonalities and variability • Displayed as one monolithic model • Widely used in the area of SPLE • Multiple derivations and extensions available
Decision Model and Notation	• Designed to work in conjunction with business processes (BPMN) • Focus on the process of decision making ignoring software structure information
Orthogonal Variability Model	• Focus on variability information ignoring software structure • Loosely interconnected variation points (via constraints) with dedicated options and selection constraints for each variation point
Common Variability Language	• Designed to integrate variability information into arbitrary realisation models such as UML • Decentralised variability specification (VSpecs) as overlay in other models • Variability model as a composited VSpec tree

To allow getting an overview of the described four basic modelling methods, Table 3.2 lists the major differences and philosophies the modelling strategies are based on.

Basically, *feature models* can be used to model a software family with its common and variable software parts as a whole. The monolithic, tree-structured approach and wide use in the area of PLE makes it unique to the other models.

Decision models in a variability context concentrate on the derivation process of concrete products out of a product line. The underlying software structure is mainly irrelevant for these models.

Orthogonal Variability Models allow the variability modelling of orthogonal variability dimensions. This tackles the problem of exponential growth of feature models when considering multiple variability dimensions.

The *Common Variability Language* allows the integration of the variability modelling process into other modelling steps such as the design of class diagrams using UML. Variability information can be integrated as decentralised Variability Specifications (VSpecs). The whole variability model is a composition of all VSpecs.

3.2 Feasibility Analysis for Partly Open Software Families

The goal of the next step is to determine a foundation for a POSF variability modelling technique. Ideally, one of the modelling methods presented in Section 3.1 will serve as the foundation.

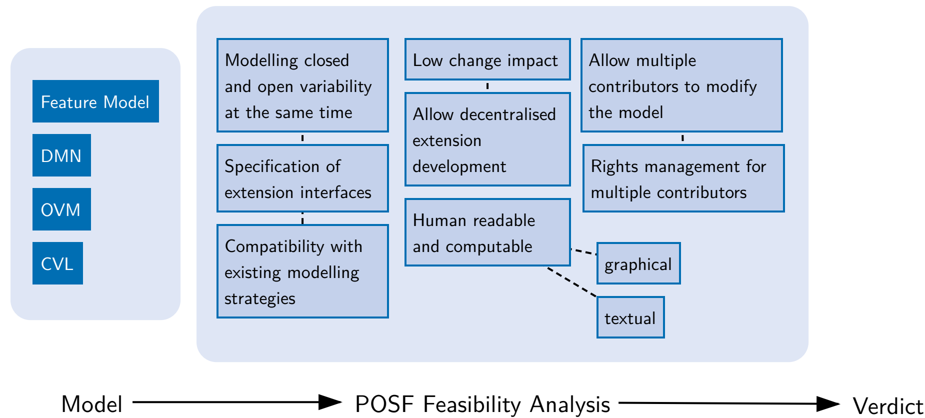


Figure 3.12 – Workflow for the models' feasibility analysis

To determine whether a modelling strategy is appropriate to be used in a POSF context or not, each model has to be analysed in regard to the derived POSF requirements. These requirements were presented in Section 2.2.1. Figure 3.12 visualises the analysis workflow.

3.2.1 Feasibility of Feature Models

Modelling closed and open variability at the same time. Feature models in their basic form do not support open variability. Such models premise on the existence of all software artefacts or features in advance. This set of features is closed and the feature model adds selection constraints by means of a tree-structure and cross-tree constraints according to that feature set.

Nevertheless, the concept of *Multi Software Product Lines (MSPLs)* [RS10] introduces a way to implement an open world concept by adding the opportunity to compose multiple product lines (represented by feature models) to one bigger product line. The composed product line is called Multi Product Line (MPL) or – within the context of software – an *MSPL*. A compositional approach allows variability, because the composed product lines can be exchanged. Hence, feature models can be exchanged to add *open* variability.

Finally, feature models do not support open variability out of the box. However, feasible extensions can be designed to enable closed and open variability modelling at the same time. The idea of MSPLs might serve as a prototypical concept.

Specification of extension interfaces. Because of the missing openness of feature models, there is a lack of interfaces. Thus, there is no specification for extension interfaces available.

This is a lack within the concept of MSPLs, too, as there is no formal, conceptual specification available by means of which feature models or product lines could be composed. The composition

using MSPLs is done by flagging features of the parent model to require a child-feature-model with a specific name. There is no further specification or semantics-based approach available.

Compatibility with existing modelling strategies. There are many extensions for feature models available. Modifications that have to be done to fully support POSFs would require another extension, which ideally does not interfere with the basic feature model and further extensions.

Low change impact. Usually, feature models cannot guarantee a low level of change impact. Even small changes could lead to a configuration deadlock or invalidation that reduces the variant space of a product line. A low change impact level can be forced by the encapsulation of model parts by forbidding cross-tree constraints outside of that encapsulation. Unfortunately, feature models do not support this yet.

Allow decentralised extension development. As there is no openness supported by feature models yet, developing extensions for such models requires access to the whole model. On the other hand, the concept of MSPLs allows the named, decentralised, autonomous extension development *to some extent*. The major limitation is the insufficiency of interfaces, because they require complete domain knowledge yet.

Human readable and computable. Whereas the human readability of visual models can be assumed, the computability is not sure in every case of a model. In case of feature models, this computability is given, because of their tree-structure with added constraints. Constraints are logical expressions that can be computed by a Satisfiability (SAT) solver. Tree-structures are graphs and, thus, can be computed using graph algorithms.

Additionally, there are proofs that show the computability of feature models. For instance, the *extFM*[2] project performs calculations over feature models. For example, quality assurance measurements can be performed using this tooling project [Gol13].

Allow multiple contributors to modify the model. The contribution to a feature model by multiple parties is not conceptually respected yet. Nevertheless, MSPLs do allow an interference-free co-operative work on a feature model because of the model separation. This concept could be adapted for POSFs as well.

Rights management for multiple contributors. There is no rights management for feature models yet, as the contribution by multiple parties is conceptually not respected. However, to support rights management, a restriction needs to be added, that permits groups of selected users to view or modify a fraction of the whole model only.

Table 3.3 summarises the feasibility analysis for feature models as a POSF modelling strategy.

3.2.2 Feasibility of Decision Models

Modelling closed and open variability at the same time. Decision models can be designed both ways, closed and open. The closed approach has a fixed set of possible decisions that guarantee a variability selection. In this case, decision tables (or other forms of decision logic) are complete. Open decision models cannot guarantee a decision as not every premised situation is respected. Hence, there is room for extensions that fill this premise gap.

[2] extFM website on GitHub: `https://github.com/extFM`

Table 3.3 – Feasibility overview of feature models; tick: passed/supported, cross: failed/not supported, circle: partly passed/limited support

Requirement	Passed	Comment
Modelling closed and open variability at the same time	◎	Partly as Multi SPL
Specification of extension interfaces	✕	
Compatibility with existing modelling strategies	✔	
Low change impact	✕	
Allow decentralised extension development	◎	Partly as Multi SPL
Human readable and computable	✔	
Allow multiple contributors to modify the model	◎	Partly as Multi SPL
Rights management for multiple contributors	✕	

Specification of extension interfaces. The extension of a decision model is limited to filling the decision logic gaps. Hence, there is some kind of restriction that forms an extension interface. The fillable gaps can be specified.

Compatibility with existing modelling strategies. Using open decision models harms the decision guarantee that might be required by the underlying product line. In consequence, compatibility with existing modelling strategies could be affected. The modelling approach itself is compatible with closed decision models, as there have not been made any conceptual changes.

Low change impact. The change impact within decision models is potentially high, because the model's focus is the process of decision making in a variable environment. The actual impact of one decision is complex to evaluate by analysing the decision model. Hence, changes that affect a decision can hardly be estimated.

Allow decentralised extension development. Decision modelling relies on the knowledge of existing premises and possible consequences. The decision logic that connects the premises with corresponding consequences is modelled within the decision model. A decentralised extension development would require knowledge of the premises, consequences and the existing logic regarding these premises and consequences. Hence, if these information is provided, a decentralised extension development is supported. In any other case, decentralisation is not feasible.

Human readable and computable. Similar to feature models, visual models such as a decision model can be assumed as human readable. Decision models are computable as well, because a complete conversion to logical expressions is possible. This can be computed using a SAT solver.

Allow multiple contributors to modify the model. Plus: Rights management for multiple contributors. As a decentralised development is only feasible to a limited extent and a low change impact cannot be guaranteed, the contribution by multiple parties is limited, too. This corresponds to the need of a rights management. Access can only be limited to the reasonable extent an external co-worker would need to contribute.

Table 3.4 summarises the feasibility analysis for decision models as a POSF modelling strategy.

Table 3.4 – Feasibility overview of decision models; tick: passed/supported, cross: failed/not supported, circle: partly passed/limited support

Requirement	Passed	Comment
Modelling closed and open variability at the same time	✔	
Specification of extension interfaces	◎	By means of logic restrictions
Compatibility with existing modelling strategies	✕	
Low change impact	✕	
Allow decentralised extension development	✕	Full knowledge is required
Human readable and computable	✔	
Allow multiple contributors to modify the model	✕	
Rights management for multiple contributors	✕	

3.2.3 Feasibility of Orthogonal Variability Models

Modelling closed and open variability at the same time. One key characteristic of OVMs is the ability to visualise multiple orthogonal variability assets at the same time and within one model. This is realised by their open structure without the necessity to cross-reference each single variation point diagram with each other to format one whole model or graph. This open structure allows a certain degree of openness of the offered variability. New variation points can be added without respecting the existing structure.

In consequence, modelling open variability is possible in a way, so that the model does not forbid adding additional variability. Hence, the modelled variability cannot be assumed as closed and fixed.

Specification of extension interfaces. Besides the fact that openness can be modelled as described above, there are no notation elements available to model extension interfaces. One possible attempt to get to some sort of interface would be the introduction of open implications and exclusions. This leads to the introduction of relations between existing variation points and not yet existing ones. This concept is similar to the composition principle of MSPLs.

Compatibility with existing modelling strategies. As OVMs allow adding supplemental variation points, these models already support to model open variability. Nevertheless, to support all of the enumerated requirements, modifications to the model are required. From this perspective, those changes should not require fundamental conceptual changes. That is why the compatibility of an *open* OVM with a *finite* OVM can be assumed. Compatibility with other models are not necessarily given. For instance, while finite OVMs can be transformed to feature models and vice versa (similar to the car infotainment example above), this might not be feasible with open OVMs.

Low change impact. The assumable change impact can be analysed from two perspectives. First, the impact of a model change on the referred software. Second, the impact of a model change on the model itself.

An OVM allows modelling orthogonal variable assets with a reduced modelling effort compared to other models, such as feature models. Thus, adding variability means adding an additional

variation point to the model. Those orthogonal variability assets may have a massive impact on the actual software, whereas the impact on the model is low.

The low change impact that is required to work with POSFs focusses on the model's change impact. Hence, modelling variation points in an OVM has a low change impact as long as cross-references (implications and exclusions) are not involved. Model changes that affect those references have an impact on the referenced model elements.

Allow decentralised extension development. Extending an OVM externally is possible as far as no cross-references are involved. Otherwise, a decentralised extension development requires additional notation elements for an OVM to introduce extension interfaces.

Human readable and computable. An OVM can be considered a set of (via cross-references partly interconnected) variation points. Each variation point is a two-layered tree-structured graph. These tree-structures can be computed using existing graph algorithms similar to the computability of feature models. Assuming a transformation method to generate a feature model from an OVM would allow the same computability as that of feature models. Such a transformation may require the introduction of additional abstract features to mount each variation point tree into the common feature tree.

Allow multiple contributors to modify the model. Allowing multiple contributors to modify the model requires the support for decentralised modifications and calls for a low change impact. Both is given with OVMs under certain circumstances that were named beforehand. Basically, OVMs support a decentralised modelling approach and a low change impact regarding the model, but cross-references, such as implications and exclusions, may interfere with that. Finally, allowing multiple parties to contribute to one model requires restrictions regarding cross-references.

Rights management for multiple contributors. As this is the case with all the presented modelling approaches, OVMs do not support a user rights management out of the box. To implement a rights management for OVMs, a per-variation-point access approach could be used. Therefore, a model can be interpreted as set of variation points. For each user, a subset could be defined that describes which variation point may be viewed or edited by the user.

Table 3.5 summarises the feasibility analysis for Orthogonal Variability Models as a POSF modelling strategy.

3.2.4 Feasibility of the Common Variability Language

Modelling closed and open variability at the same time. The Common Variability Language is used to integrate variability modelling into software realisation models such as UML. The CVL offers model elements that are called Variability Specification (VSpec). These specifications allow the partial integration of variable elements into other models. For instance, VSpecs can be added to a UML class hierarchy to add variability information referred to that part of the software that is modelled by the base class diagram.

As those VSpecs directly refer to the modelled software part, the variability refers to a closed set of features. Openness cannot be visualised using this part of the CVL concept.

The variability model as a whole is the composition of VSpec trees to a VSpec tree set. Each VSpec is decoupled from other VSpecs. Only logic elements that could implement implications

Table 3.5 – Feasibility overview of orthogonal variability models; tick: passed/supported, cross: failed/not supported, circle: partly passed/limited support

Requirement	Passed	Comment
Modelling closed and open variability at the same time	✔	
Specification of extension interfaces	✕	
Compatibility with existing modelling strategies	✔	
Low change impact	✕	
Allow decentralised extension development	✔	If cross-references are not involved
Human readable and computable	✔	
Allow multiple contributors to modify the model	◎	Based on the involvement of cross-references
Rights management for multiple contributors	✕	

and exclusions interconnect the available Variability Specifications with each other. Hence, additional VSpecs can be added retrospectively. Regarding this, CVL models allow openness.

Specification of extension interfaces. There are no notation elements that allow the specification of interfaces. The openness of CVL models is limited to the possibility of adding VSpecs retrospectively. Interface specifications would require model changes. Developing such an extension could be challenging, because the variability model is formed loosely based on the existing (non-open) VSpecs. Thus, interface specifications require elements next to those VSpecs. The idea behind CVL to allow the integration of variability into other models, such as UML diagrams, should be kept. Hence, the new model elements have to be embeddable into existing base realisation models.

Compatibility with existing modelling strategies. The CVL is designed to get implemented within other models. Thus, these models are made to be compatible with other existing modelling strategies. This design concept guarantees the cross-model compatibility. When adding extensions that support an open interface specification the possibility to integrate CVL diagrams into other models must not be harmed. In consequence, the compatibility should be kept, too.

Low change impact. The impact of changes within a CVL model relies on the constraint expressions that are part of the VSpec trees. In case these logic expressions remain unchanged, a low change impact is given by means of the decentralised modelling concept. Otherwise, the change impact is potentially high.

Allow decentralised extension development. The external extension development in the CVL case is coupled with the modelling opportunities of the underlying model. If this model supports an external extension development, the CVL supports adding variability externally, too. This is only limited by the structural dependencies formed by the logical expressions or constraints within the VSpecs.

Human readable and computable. The variability model as a whole is formed as a set of VSpecs. Each VSpec may encompass constraints as logical expressions. Similar to feature models, each VSpec can be visualised as a tree-shaped graph. Thus, graph algorithms can be

applied to compute VSpecs. Logical expressions can be computed using SAT solver systems. Furthermore, the constraints form relations between the VSpec trees. Hence, the interconnected trees can be interpreted as a larger graph, where graph algorithms apply, too.

Allow multiple contributors to modify the model. The decentralised extension development is supported as described above if the underlying realisation model supports it. That allows multiple contributors to add extensions to the model. The lack of interface specifications limits that collaboration attempt.

Rights management for multiple contributors. There is no rights management supported via the CVL concept. If the underlying model supports rights management, the VSpec trees may be covered by that, too.

Table 3.6 – Feasibility overview of the Common Variability Language; tick: passed/supported, cross: failed/not supported, circle: partly passed/limited support

Requirement	Passed	Comment
Modelling closed and open variability at the same time	×	
Specification of extension interfaces	×	
Compatibility with existing modelling strategies	✔	
Low change impact	×	
Allow decentralised extension development	◎	Depends on the underlying model
Human readable and computable	✔	As a set of VSpecs
Allow multiple contributors to modify the model	◎	Depends on the underlying model
Rights management for multiple contributors	◎	Depends on the underlying model

Table 3.6 summarises the feasibility analysis for CVL models as a POSF modelling strategy.

3.2.5 Feasibility Analysis Verdict

Based on the given analysis for each modelling method presented in this chapter, a conclusion should summarise the results and extract a design decision for Chapter 4. In this next chapter, a concrete, feasible modelling concept shall be built regarding variability modelling for Partly Open Software Families.

To formulate a conclusion of the preceding analysis, a brief and objectively comparable overview is given in Table 3.7. This table consolidates the analysis results of Table 3.3 to 3.6. A first look at this consolidated table shows that none of the presented modelling methods fulfill all defined POSF requirements of Figure 2.7. This is the first conclusion:

None of the presented variability modelling techniques fulfill all given requirements.

The bottom line of the table counts all ticks, circles and crosses per column or model. A tick means the model satisfies the given requirement, whereas a cross is given if the model does not support the required characteristic or behaviour. The circle indicates that the analysed model satisfies the requirement in a limited way. Either there is an indirect way to support the desired behaviour or characteristic, or the requirement is fulfilled under specific circumstances.

Table 3.7 – Consolidated feasibility overview for all presented modelling techniques; tick: passed/-supported, cross: failed/not supported, circle: partly passed/limited support

Requirement	Feature Model	Decision Model	Orthogonal VM	Common VL
Modelling closed and open variability at the same time	◎	✔	✔	✕
Specification of extension interfaces	✕	◎	✕	✕
Compatibility with existing modelling strategies	✔	✕	✔	✔
Low change impact	✕	✕	✕	✕
Allow decentralised extension development	◎	✕	✔	◎
Human readable and computable	✔	✔	✔	✔
Allow multiple contributors to modify the model	◎	✕	◎	◎
Rights management for multiple contributors	✕	✕	✕	◎
	2x ✔ 3x ◎ 3x ✕	**2x ✔ 1x ◎ 5x ✕**	**4x ✔ 1x ◎ 3x ✕**	**2x ✔ 3x ◎ 3x ✕**

Based on these symbol counts, decision models lose that comparison with five failed requirements and three passed or partly passed requirements. All other models only have three failed requirements versus five fulfilled or partly fulfilled criteria. The best match is the OVM method with four passed criteria and just one partly passed criterion. Nevertheless, a final decision requires a more detailed look at the results. Thus, the second conclusion is:

> *Decision models have the worst match of fulfilled and partly fulfilled requirements versus failed requirements. The other models have a significantly better ratio.*

This second conclusion brings feature models, OVMs and CVLs into the focus. It would be ideal to select one of the named models to serve as a foundation to a modelling technique that can be used to model the variability of POSFs. In consequence, it is important to estimate the effort to modify an existing model to support all required characteristics and behavioural aspects.

Feature models do not support the specification of extension interfaces, do not guarantee a low change impact and offer no rights management yet. Furthermore, modelling open variability is just offered by the MSPL concept that introduces the composition of multiple feature models. This compositional concept is as well the reason for the partly satisfied requirement of a support for decentralised extension development and the model modification by multiple contributors.

To match the missing requirements, a compositional approach similar to MSPLs could be introduced and new notation elements are required to model interfaces and interface specifications. These specifications open the demand for a semantics specification next to the pure structural assets available in feature diagrams. Rights management is necessary to allow multiple different parties to contribute to a model by adding concrete extensions to the specified interfaces. The rights management could be realised as a per-feature attribute. The change impact has to be reduced as far as possible, for instance by adding conceptual rules to limit cross-tree constraints. Those constraints can be seen as the major issue regarding the level of change impact.

Orthogonal Variability Models lack the support for interface specifications, the low change impact and a rights management, too. Similar to feature models, there are new notation elements

necessary to implement interfaces. The support for semantic interface specifications is required, too. Besides the non-monolithic approach of OVMs, allowing multiple contributors to modify the model requires restrictive rules according to the cross-references. The challenging aspect is the high reliance on these cross-references. Whereas relations of feature models are separated in parent-child relations and cross-tree constraints, OVMs only support the named cross-references. Parent-child relations are restricted to the tree-structure of a feature diagram and allow an easier to determine change impact, while cross-tree constraints and OVM's cross-references are not restricted per default.

The *Common Variability Language* supports an integrated variability modelling within software realisation models such as UML class diagrams. That is why the necessary modifications to support POSFs rely on the base model. Especially the decentralised modelling is supported due to the loose binding of VSpecs but, finally, depends on the compositional opportunities of the base model. This makes it rather vague to estimate the modification effort for CVL models to satisfy all POSF criteria.

This leads to the third conclusion:

> *Feature models and Orthogonal Variability Models have a similar modification effort to create a modelling method that satisfies all POSF requirements. For CVL models, this estimation is vague due to the reliance on a base model.*

Based on these three conclusions, feature models and OVMs can be used to serve as a foundation to create a feasible POSF variability modelling strategy. The estimated effort is similar. Whereas feature models additionally require the introduction of a compositional approach similar to MSPLs, achieving a low change impact in OVMs may be challenging because of the flat referencing system compared to the more divers referencing supported within the tree structure of feature models.

As a verdict based on the analysis above, feature models can be considered the best option to serve as a POSF modelling basis. Additionally, feature models are widely used in the context of Software Product Lines.

3.3 Summary

Four modelling strategies were inspected in this chapter: feature models, decision models, Orthogonal Variability Models (OVMs) and models based on the Common Variability Language (CVL). Each of these models were presented in detail according to the scheme shown in Figure 3.1. Especially the basic modelling process and the notation principles are important to allow a feasibility analysis against the POSF requirements defined in Chapter 2.

A brief summary of all presented modelling methods allowed a first comparative view on the models according to their suitability for a use in the context of POSFs. The feasibility analysis in Section 3.2 examined each of the four models regarding the former defined requirements. The results were summarised into tables using symbols to show whether a requirement is fulfilled, partly fulfilled or failed.

The given verdict consolidated the single analysis results and evaluated them to extract a design suggestion for the next chapter. The extracted suggestion is to use feature models as a foundation to develop a new modelling concept. A rights management has to be introduced, a compositional approach similar to that of MSPLs shall be introduced and elements should be included that allow a viable interface specification to support multiple, external model contributors.

Chapter 4 will use this suggestion and the results of the former chapters to build a variability modelling concept for Partly Open Software Families that suffices the defined requirements. Additionally, a prototypical tool implementation will be given supporting modelling variability for POSFs based on the introduced modelling concept. The intention of this tool is to proof feasibility of the modelling concept and allow further evaluation by means of a case study later in this thesis.

4 Creation of a Feasible Concept

The former chapter presented four different variability modelling techniques and extracted the suggestion to use feature models as the foundation to create a new modelling concept for Partly Open Software Families. This suggestion is based on the analysis of the presented models regarding their support for POSF requirements. These criteria were elaborated in Chapter 2.

This chapter combines the results of the chapters above to create a new and feasible variability modelling concept for POSFs. Therefore, Section 4.1 presents the necessary modifications of feature models to support POSFs.

Furthermore, this chapter describes the conception and implementation of a prototypical tool that shall proof the feasibility of the created model in Section 4.2.

4.1 Creation of a Feasible Variability Modelling Concept

The intention of the former chapters was to seek for an existing modelling strategy that can be extended to support the design of POSFs. The major advantages of this approach are the avoidance of unnecessary redundancy and the usage of a modelling mechanism that already exists and is used in practise. A completely new model would have to proof itself in practise, for which there is no evidence yet.

Chapter 3 proposed to use feature models as a foundation to develop modelling extensions to support POSF specifics. Taking this suggestion into account, according to Table 3.3, the following changes to the model are required:

1. There is no sufficient way to model an open software structure using feature models yet. To enable this, new notation elements that support the introduction of interfaces are required. Section 4.1.1 tackles this issue.

2. An interface usually comprises a specification of requirements new extensions shall satisfy. Therefore, the notion of structural specifications needs to be defined. Ideally, a graphical representation of the model supports adding concrete extensions based on the interface specification as well. This is useful for external contributors to model extensions. Furthermore, the support for concrete extensions inside of the model allows the transformation of a POSF feature model into a normal feature model – e. g. for compatibility reasons. Section 4.1.2 introduces a way to support both structural specifications and concrete extensions.

3. In contrast to MSPLs, interface specifications should comprise semantics information, too. Feature models follow a structural approach, which does not allow the inclusion of explicit semantics information. Section 4.1.3 proposes ways to solve this issue.

4. One of the key aspects of POSFs is the possibility to collaborate in building a software. The allowed openness is one aspect that enables this. The second is the need of rules for regulating of what each collaborator is allowed to see and permitted to modify. Thus, an appropriate rights management is required, which will be introduced in Section 4.1.4.

4.1.1 Introduction of Interface Notation Elements to Feature Models

To tackle the first issue – the lack of openness support – a new type of parent-child relation is introduced. Until yet, feature models support the following types of parent-child relations:

- Binary relation
  - ...for mandatory features
  - ...for optional features
- Group or set relation
  - ...as an OR group (at least one feature has to be selected)
  - ...as an XOR group (exactly one feature has to be selected)
  - or as a group with lower and upper bound values to regulate the cardinality

Interfaces that enable openness can be introduced as another type of parent-child relation in addition to the above mentioned ones. To do so, a feature can be turned into an *interface* or *open feature* by adding a dock connection onto it. This connection specifies that this feature supports the hooking of concrete extensions to that interface. The interface specification and the hooking of concrete extensions is subject to the next section.

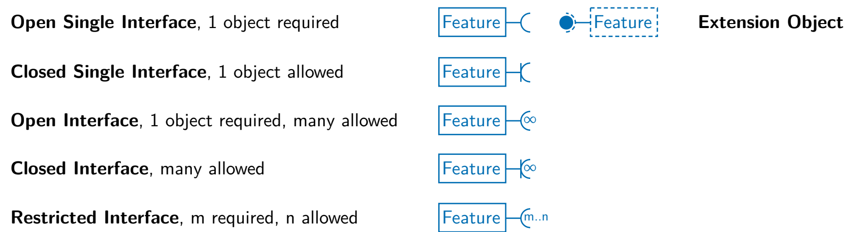


Figure 4.1 – Proposed POSF notation elements as an extension to the FODA notation for feature models

Figure 4.1 shows a proposal of how such a dock connector could be denoted. The basic notation is inspired by the interface notation used in conjunction with components in UML diagrams. The semantics of this element in UML and POSF feature models should not be mixed up, but the common intention of interfaces and hooks leads to this design decision. Furthermore, the proposed dock connector (lollipop notation) can easily be distinguished from the remaining parent-child relations. It is crucial to know that there is no actual relation between the UML component interfaces and feature model interfaces. Whereas these interfaces in UML are used to model communication processes, the proposed interface notation in feature models is used to model software structure.

The dock connector as proposed can be annotated with cardinality information of how many extensions are allowed to be hooked. Thus, it could be necessary to specify that exactly one extension is required to be installed. To serve an example, a data management application could need a data source provider, which could be a database or files in a filesystem. The data source provider can be realised as an interface-extension relation. As the application may need just one particular provider to work, this is one case where requiring an explicit number of extensions is useful and necessary. Thus, there should be a general support for cardinalities with a lower and an upper bound. The proposed notation for cardinality is given by Figure 4.1.

The default case (without further cardinality information in the model) describes that exactly one specific extension is required to implement the interface. This is denoted as a half circle. To mark an interface as optional, a “blocking” line is proposed. To allow infinitely many extensions respectively remove upper bound cardinality restrictions, the cardinality can be denoted using the ∞ symbol. In any other case with specific lower and upper cardinality bounds, this can be denoted using the usual notation *[m..n]*.

The interface specification is given by a hook for the dock connector (lollipop notation). This is shown by the *Extension Object* element in the figure. The dashed half circle is proposed to avoid possible confusion with the mandatory feature relation. The interface specification starts with the same root feature name as the name of the interface feature. During the concrete modelling of extensions, the name of this extension root feature can be exchanged, e. g. by the extension’s name.

4.1.2 Differentiation of Interface Specifications and Concrete Extensions

Modelling openness inside a feature model means – on the one side – to allow the notation of interfaces and their specifications. On the other side, the graphical notation of a feature model should allow the integration of concrete extensions that are hooked into a model. This comes across as the realisation of an abstract interface into a concrete extension. Especially during the modelling process of external extensions, it could be useful to integrate them directly into the underpinned feature model basis. Additionally, supporting the inclusion of concrete extensions into a model allows the transformation of a POSF feature model (with open variability) into a usual feature model (without open variability).

To allow modelling both interface specifications and concrete extensions inside of one model, an explicit distinction is required. To do so, the interface relation is proposed to be realised as a specification relation *or* as a set relation of concrete extensions.

While interface specifications use the lollipop notation, concrete extensions are proposed to use an arrow that points to the implemented interface feature.

Specification of an Interface. In addition to modelling an interface feature by using a docking connector, the interface has to be specified in its structure and semantics. The semantics shall be handled by one of the possible solutions given in Section 4.1.3. The structure shall be modelled similar to an ordinary feature model. Hence, there is an extension’s root feature which comprises child features with modelled binary and group relations. Cross-tree constraints are allowed, too. The name of the extension’s root feature equals the parent interface feature. This name may change while modelling a concrete extension. The modelled interface specification serves as a basis for a concrete extension. Thus, it acts as a stem for further modelling work. To distinguish an interface specification from concrete features of the model inside the graphical notation, it is useful to use a dashed rectangle instead of a solid border.

Notation of Concrete Extensions. The set of concrete extensions potentially contains infinitely many concrete extensions. To differentiate each extension from others, unique names (denoted as the extension’s root feature name) should be used here. During a configuration process, on whose end there is a concrete product configuration, the cardinality given in the interface specification has to be obeyed to keep the model valid. In consequence, disrespecting the cardinality invalidates the model. A valid extension has to carry over the feature stem given by the specification. However, additional features are explicitly supported!

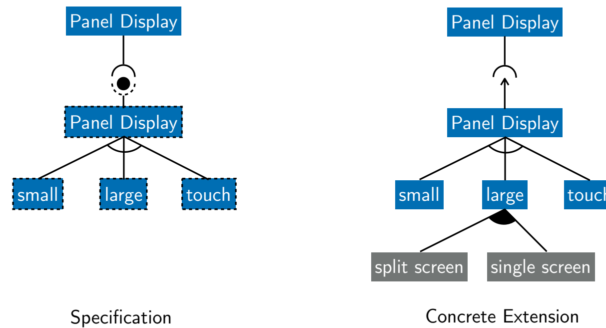


Figure 4.2 – Notation of interface specifications and concrete extensions

Figure 4.2 shows a notation example for the mentioned interface specifications and concrete extensions.

4.1.3 Addition of Semantic Information to Interface Specifications

Feature models natively only support the integration of structural information. One distinguishing characteristic of POSF-supported feature models in contrast to MSPLs shall be the possibility to validly specify extension interfaces. However, to support adding semantic information, a requirement specification has to be annotatable inside the structural feature model. The formalisation of the requirements is done using a predefined methodology.

To realise this requirements specification, there is no one-fits-all solution available. A "perfect" specification could be done using formal logics. However, logical formalisation of requirements is coupled with a relatively high effort, which may be infeasible in practise. Apart from that, limiting the interface specification to a pure name matching of the feature names of the model structure is too loose in its level of detail and there is no real semantics information included at all.

In consequence, there is no feasible alternative to a tradeoff decision. A requirements specification is needed that can be formalised with a moderate effort and with a level of specification detail that is sufficient to ensure that developing an extension against an interface leads to a valid extension.

There are multiple levels of specification detail for those semantics formalisation methods.

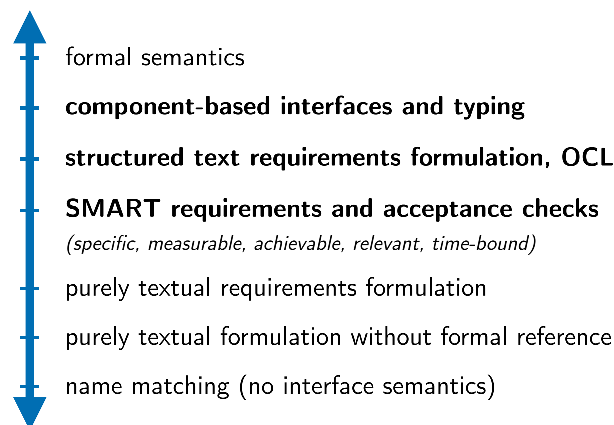


Figure 4.3 – Levels of different semantics notation approaches

Figure 4.3 visualises a range of different methods as a scale from the mentioned *formal semantics* as the highest level of detail to *name matching* as the lowest level. Pure textual requirements formulations with or without formal references to the actual software and measurable criteria are difficult to evaluate reliably. That means, the determination whether a requirement is fulfilled or not could depend on subjective interpretation. Formal semantics specification, on the other hand, often is too expensive to realise. Hence, a balance between specification effort and level of specification detail is required.

A reliable and feasible requirements specification can be considered sufficient if it allows an exact interpretation of what the described software should be able to accomplish and if it allows a clear determination of fulfillment – meaning that it is unambiguous to determine, whether a requirement is satisfied or not.

SMART Requirements and Acceptance Checks. The least formal but still sufficient method of Figure 4.3 to specify an interface semantically is the formulation of requirements using the *Specific, Measurable, Achievable, Relevant and Time-Bound (SMART)* rule. There are several similar interpretations of the acronym SMART in this context, but basically, this can be interpreted in the following way based on the goal formulation rules by [Bog05]:

- ***S****pecific:* A specific goal/requirement is formulated as exact and explicit as possible. It should be unmistakable and clear what is meant in an effective way. Ideally, the formulation is as short as possible. Instead of “a sufficiently short boot-up time”, the goal should be “to ensure a boot-up time from cold start to login screen in at most ten seconds by loading the user interface with the highest computing priority”.

- ***M****easurable:* A measurable goal guarantees the reliable determination, whether a requirement is satisfied or failed. The given boot-up time example demonstrates that. It is not determined, what a “sufficiently short time” is, but ten seconds can be measured.

- ***A****chievable:* Achievable goals ensure that a requirement can be realistically fulfilled, which is necessary to provide valid extensions to a POSF. Unachievable requirements could potentially invalidate a POSF model completely.

- ***R****elevant:* If a goal or requirement is relevant, it is worth satisfying. This is less important to requirements specifications rather than to project goals, but – nevertheless – they could lead to motivation for developers to satisfy or even overfulfill a requirement.

- ***T****ime-Bound:* Time-bound goals are necessary to specify the latest moment of goal measurement to decide, whether the goal is fulfilled or not. In terms of requirements for a software interface, this element of the SMART rules is less relevant. However, the above specified boot-up time is some kind of time-binding, but not every requirement needs performance assertions.

The key of formulating requirements using the SMART rule is to build unmistakable and valid requirements that can be measured using an *acceptance check*. Definition 4.1 serves an explicit description of what acceptance checks are.

Acceptance checks allow an at least mostly automated requirements check. The goal is to describe the semantics of a software interface using measurable requirements that can be implemented into automated test cases, which are able to determine the level of fulfillment of the interface.

Definition 4.1: Acceptance or Conformance Check

"Acceptance testing is about validating the software product against user requirements. Acceptance test models are generated from the user requirement specifications. There are two steps in acceptance testing: Alpha & Beta. They usually consist of the same set of test cases, but they are applied at different locations and times. Alpha is performed on the development platform before the deployment of the product; while Beta is executed on the target platform at the user site during the deployment of the product and mostly performed by the user."[MK14]

Structured Text (Constrained Natural Language) Formulation of Requirements and OCL. The method of formulating requirements using the SMART criteria and acceptance checks can be lifted to a more formal notation. Instead of a free prose formulation of a requirement, structured language and patterns can be used to normalise the formulations. Based on this, the Object Constraint Language (OCL) can be used to parse these requirements formally on the model level. The OCL is used to introduce constraints into models that are beyond the expressiveness of the UML realisation model. While UML models – similar to feature models – only express structural software information, semantic data can be added by means of those OCL constraints. Thus, they can be used to introduce measurable requirements to a structural model, which is not limited to UML models. [RG99]

Constraints based on the OCL allow to express the following elements:

- *Invariants:* conditions that are required to be steadily valid.
- *Pre and post conditions:* constraints that are valid before or after a specific behaviour.
- *Initial and derived values:* constraints for initial states of a software and resulting values.
- *Guards:* constraints that specify, whether an operation can be performed or not as they need to be valid or true first.

These elements can express the (semantic) requirements alongside the concrete behaviour of a software operation.

To realise an automated checking process, more detailed information about the realisation of extensions is necessary, which – finally – leads to some kind of acceptance checks again. Thus, conformance checks can be written based on the modelled OCL constraints to automatically check a concrete extension against the interface specification.

Component-Based Interfaces. A methodology that is near the realisation modelling is to semantically specify interfaces of POSF models similar to interfaces of Component-Based Software Engineering (CBSE). CBSE is a programming paradigm next to – for instance – Object-Oriented Programming (OOP). Different software components interact with each other using interfaces, while the actual realisation of each component is black-boxed and, thus, invisible to other developers. The interfaces include the complete interaction information that is necessary for an implementation. This comprises semantics information as well.

To formulate the interface specifications, an Interface Description Language (IDL) is used. There are various kinds of IDLs which are related to a specific context or domain. Just to name a few: When developing for Android, for instance, the IDL called *AIDL* is used, which is a

language based on Java. In the context of web applications, the *Web IDL* can be used. A rather independent solution is to use the *OMG IDL* proposed by the Object Management Group Inc. or the slightly extended *Microsoft Interface Definition Language (MIDL)* from Microsoft.

The commonality of all of these IDLs is the formal formulation of interface requirements, often including a strong typing support. This facilitates the conformance checking of extensions and ensures that concrete extensions are designed as intended.

When to Use One of the Presented Semantic Specification Methods? The major difference between the presented semantic description methods is the level of detail each of the methods can deliver. The more detailed a description is, the closer it is to the actual realisation level. The component-based approach utilises a programming-specific IDL. In contrast to this, the formulation of SMART requirements is not specific to any realisation strategy.

There is no restriction to decide for only one method to formulate an interface specification. They may be mixed depending on the required level of detail for each interface. Generally, the highest level of specification detail is desirable, but the effort to do this specification work has to be estimated and considered when deciding for a strategy.

As a rule of thumb, this thesis suggests to use SMART requirements whenever an interface can be sufficiently specified using this method, because it delivers a balanced tradeoff between specification detail and effort.

4.1.4 Introduction of a Rights Management

One of the key intentions of Partly Open Software Families is the support for multiple contributors. Hence, there has to be a controlling layer that implements a rights management using access rules. This rights management shall specify who is able to modify the model, which part of the model is modifiable and what is visible but immutable by a user. Limiting visibility can be desired due to potential business secrets.

To implement such a rights management system, the following three components are necessary:

1. A *user management* system is required to uniquely identify different contributors.
2. To define what users shall be able to see or modify inside the model, certain *rules* are necessary. They specify the actual effect as soon as a user is permitted to see or modify an element of the model.
3. Finally, a *user-feature mapping* is required that assigns users to features to grant them access.

Additionally, but separately from the list above, a *controlling software* is required to implement these components by holding the user management, observing the user-feature mapping and ensuring that the given rules are obeyed.

The realisation of such a rights management can be done in a similar way as it is known from filesystems. Thus, *managing users* can be done by keeping users organised in user groups and alongside with the access credentials inside of a data base. A user identification (log-in) is required to access the model from each user's perspective. Managing users in groups has the advantage of granting rights in a group-wise way. For instance, to allow some users full access to the model, a model maintainers group is supposed to be added, such that the group gets the permission rights. Adding a new model maintainer just requires to add a new user to this model maintainers group. There is no need to update the user-feature mapping afterwards.

The *user-feature mapping* is an assignment of users or user groups to features. It has to be possible to assign multiple users and user groups to a feature without any restrictions. To avoid rights conflicts, the defined user rules have to be designed as opt-in rules, which grants access permissions instead of denying them.

The *rules* state what a user or user group assignment to a feature causes. As just mentioned, by default, the whole model is invisible and immutable to a user, except there is a user-feature assignment and a rule that grants permission to make parts of a model visible or modifiable (opt-in principle). Assignments can be done to features. Thus, the rules have to specify, which part of a model, relative to this assignment feature, shall be visible or modifiable. Therefore, the following rules are proposed:

1. The visibility rules state what part of the model is visible and modifiable to all explicitly assigned users and to users of assigned user groups.

2. The feature, the user/user group is assigned to (assignment feature), is visible but immutable as well.

3. All parent features of the assignment feature are visible but immutable, meaning the path to the root feature is visible, including the parent-child relations.

4. Considering the assignment feature as a root feature of a sub-tree structure of the whole feature tree, this sub-tree is entirely visible including all sub-features and parent-child relations.

5. All cross-tree constraints are visible that contain a reference to at least one feature that is visible according to the rules above. The modification of cross-tree constraints is only allowed if all referenced features are modifiable to the user. An exception are implications, such that in this case the condition (left expression of the implication) exclusively has to comprise only features the user has modification rights to.

6. Modifying the model is solely permitted according to the sub-tree of the assignment feature. The assignment feature itself is immutable.

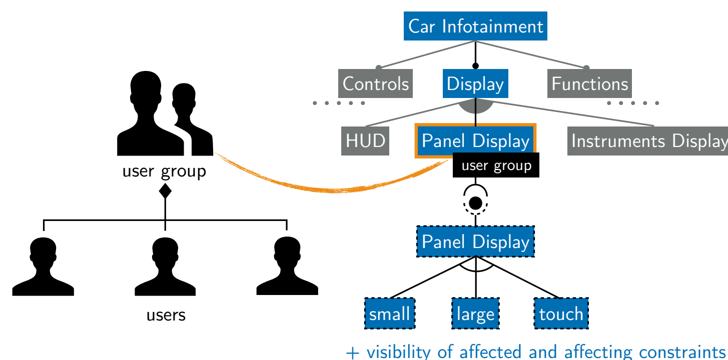


Figure 4.4 – Visualisation of the rights management rules; blue features are visible to the users of the user group, grey features are invisible. The orange bordered feature is the assignment feature. All child-features of the assignment feature are modifiable. Everything else is immutable.

Figure 4.4 illustrates the effect of these rights management rules. The user group is assigned to the feature *Panel Display*. Hence, each user of this user group is allowed to see that *Panel Display* feature as well as the path features to the root (*Display* and *Car Infotainment*). Furthermore, the sub-tree of the assignment feature is visible (*Panel Display* specification, *small*, *large* and *touch*). If there are any cross-tree constraints that affect one of the visible features, they are visible, too.

4.1.5 POSF Modelling Requirements Check

Table 4.1 – Fulfillment of the POSF requirements – comparison of feature models and the new POSF-enabled feature models

Requirement	Feature Model	POSF-enabled FM
Modelling closed and open variability at the same time	◎	✔
Specification of extension interfaces	✕	✔
Compatibility with existing modelling strategies	✔	✔
Low change impact	✕	✕
Allow decentralised extension development	◎	✔
Human readable and computable	✔	✔
Allow multiple contributors to modify the model	◎	✔
Rights management for multiple contributors	✕	✔
	2x ✔ 3x ◎ 3x ✕	7x ✔ 0x ◎ 1x ✕

To evaluate whether the above mentioned changes are sufficient to be used as a POSF modelling strategy, the new model should be checked against the former stated model requirements. Therefore, Table 4.1 serves as an overview. Additionally, the results of Table 3.3 are integrated to allow a direct comparison of the original feature model and the new POSF-enabled feature model.

The introduction of *interfaces* now fully allows modelling open next to closed variability in one model. The MSPL approach of feature models is not sufficient here.

An extension specification alongside with the interfaces is not supported in the original feature model. The introduction of structural and semantic *interface specifications* helps to satisfy this requirement.

Allowing multiple parties to work with one model requires some controlling elements inside the model. Especially a sufficient rights management can help to fulfill these requirements. In consequence, the new POSF-enabled feature models allow a decentralised extension development (using the interface specifications and user access rules) and, based on that, allow multiple contributors to work with one model simultaneously.

What is missing yet is a low change impact. Originally, feature models cannot guarantee a low change impact, as cross-tree constraints may lead to an unexpected impact on the whole model even when performing minor changes only. The worst case would be an entire invalidation of the model as a modification could cause a “deadlock” during the configuration process to derive a concrete product of a product line. This problem has not been tackled yet by the new model

proposal. Hence, even minor changes to a POSF model could lead to a potentially high impact on other parts of the model. This problem is tackled in Chapter 5 including a full change impact analysis and deriving measurements to reduce the change impact.

4.1.6 Model Overview and Example

To summarise the proposed modelling technique, this section gives an overview using an example in Figure 4.5. This example shows the car infotainment example introduced in Chapter 3. To demonstrate open variability, the panel display of this system shall be modelled as an exchangeable component. This could be useful if there are multiple panel display vendors available and the car should support a variety of them. Therefore, the interface requires the integration of exactly one panel display extension. The original feature model offered three panel display options to a customer: a small panel, a large one and one with touch support. When opening up the model by allowing external contributions, it should be ensured that each developed extension to that model supports the same richness in features as before. Thus, an extension has to offer the choice between small, large and touch as well. In consequence, this forms the interface specification a concrete extension has to satisfy. Additionally, the display interface should be managed by the user *display developer*. To demonstrate the extensibility of the infotainment system, the example includes a concrete extension that satisfies the specification as well.

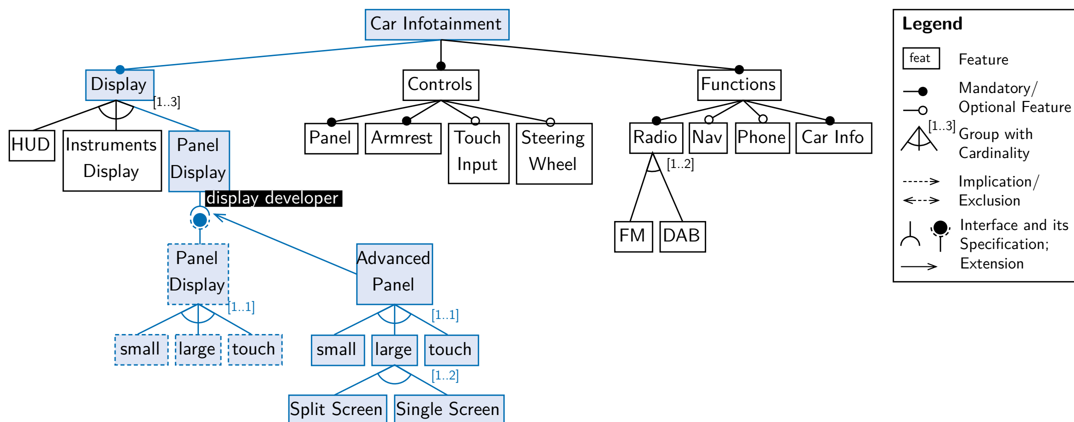


Figure 4.5 – Exemplary overview: POSF modelling based on the car infotainment example with cardinalities (cf. Figure 3.4)

In addition to Figure 4.5, the following two cross-tree constraints are given. There are two features called *touch* in the model. As a result of the unique name assumption of feature models, there is no need for a differentiation. The constraint's reference to *touch* refers to both features.

- $Nav \rightarrow PanelDisplay$
- $TouchInput \rightarrow touch$

To introduce interface semantics to each of the features given in the structural specification, the following specific and measurable requirements description (according to SMART) is given:

Panel Display The panel display offers an Liquid-Crystal Display (LCD) flat screen on top of the car's middle console. The screen has to be visible directly to the driver.

small The small screen option has a diagonally seven inches large display with an aspect ratio of 4:3 and a resolution of 640 to 480.

large The large screen option has a diagonally ten inches large display with an aspect ratio of 16:9 and a resolution of 848 to 480.

touch The touch screen option has a diagonally ten inches large display with an aspect ratio of 16:9 and a resolution of 1696 to 960, and it allows single touch point-and-click input.

One remark according to these specifications: There is no restriction to formulate distinct specifications. Nevertheless, the formulation of distinct specifications avoids potential confusion. For instance, if the display resolution in these specifications would not be fixed but a minimum resolution, each *touch* display would be a *large* display, too. This could be undesired and potentially leads to confusion and realisation conflicts.

The proposed example extends the car infotainment example of Figure 3.4. The *Panel Display* feature is modelled as an interface which requires exactly one extension (implicit cardinality *[1..1]*). The structural specification is given right below and contains the choice between a *small*, *large* and *touch* display.

A concrete extension is shown, too, called *Advanced Panel*. This panel extension obeys the specification by integrating a *small*, *large* and *touch* display and extends the large panel option by a *Split Screen* and *Single Screen* view.

To illustrate the user management system, there is a user assignment to the *Panel Display* interface feature. The user is called *display developer*. Due to this assignment, the highlighted part of the model is visible to the user including the given cross-tree constraints and the semantics specification. Additionally, he is allowed to modify the interface specification as he is assigned to the interface feature (the upper *Panel Display*) instead of the specification feature (the lower *Panel Display*).

4.2 Development of a Prototypical Tool

To prove the feasibility of the proposed modelling concept, a prototypical tooling implementation is required to perform an evaluation using a case study. The case study and evaluation is subject to Chapter 6. The following part of this chapter is used to describe the conception and implementation of the mentioned prototype.

4.2.1 Conception

The conception of a prototypical modelling tool comprises a fixation of use cases the tool shall support to perform. Furthermore, a data structure for the model has to be drafted and the user interface for the tooling has to be designed as a mockup draft. Afterwards, the technological foundation to build the tooling on should be determined, before the actual implementation can be done.

Use Case Conception. The prototypical tooling should enable a user to create a POSF model as proposed in Section 4.1. Therefore, the tool has to be capable of handling models of the proposed structure including interfaces and interface specifications. Furthermore, there should be a way to model the structure of concrete extensions. To allow collaboration, the tooling should support working in different user perspectives. Thus, the rights management should

be implemented in a way that shows how the stated rules are applied. The decentralised and external contribution to the model shall be realisable in a way, such that parts of the model (i. e. the extensions) can be exported and imported into another model.

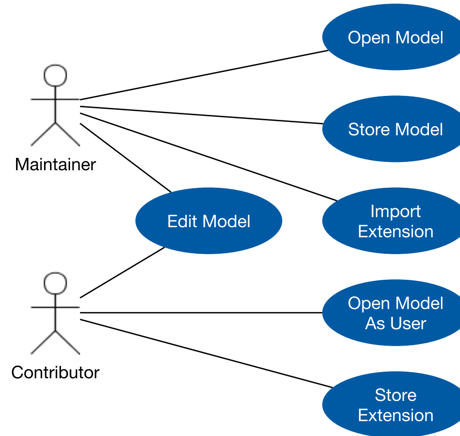


Figure 4.6 – Use cases for the prototype (UML use case diagram)

Figure 4.6 shows a *use case diagram* that formalises the mentioned scenarios the prototype shall be able to perform.

A *Maintainer* of the model shall be able to open, modify and store a POSF model. Furthermore, he should be able to import externally supplied extensions into the existing model.

An external *Contributor* has to be able to open the model by obeying the correct visibility and modifiability rules by respecting the rights management. Once the model is opened correctly, he has the chance to edit the permitted parts of the model, which comprises creating and editing concrete extensions. Furthermore, there has to be the opportunity to export a concrete extension, so that it can be imported into another model by its maintainer.

Data Structure Conception. The prototype has to handle POSF models. Therefore, a meta model is required to describe the structure of such a model.

The proposed model is based on a regular feature model and is extended by certain additional elements. That is why the data structure for the prototype is based on the feature modelling meta model shown in Figure 3.3 in Chapter 3. Figure 4.7 shows a rough UML class diagram structure of the proposed meta model.

The different aspects of that model are colour-coded. The central part of the model is the root class `FeatureModel` that comprises the data structure to handle. Furthermore, the `Feature` class is obviously crucial for modelling feature models.

The lower-left blue section of the figure models the parent-child relations of the model to realise the model's tree-structure. A relation could be binary (`Mandatory` or `Optional`), a group relation (`Set`) or describes an interface.

- `Specification` representing an interface specification and
- `ExtensionSet` comprising a set of concrete extensions

The upper-right violet part models cross-tree constraints as implications (`Imply`) and exclusions (`Exclude`). They comprise a logical expression, whereas the logical expression is tree-structured itself with feature references (`FeatureRef`) as leaf nodes.

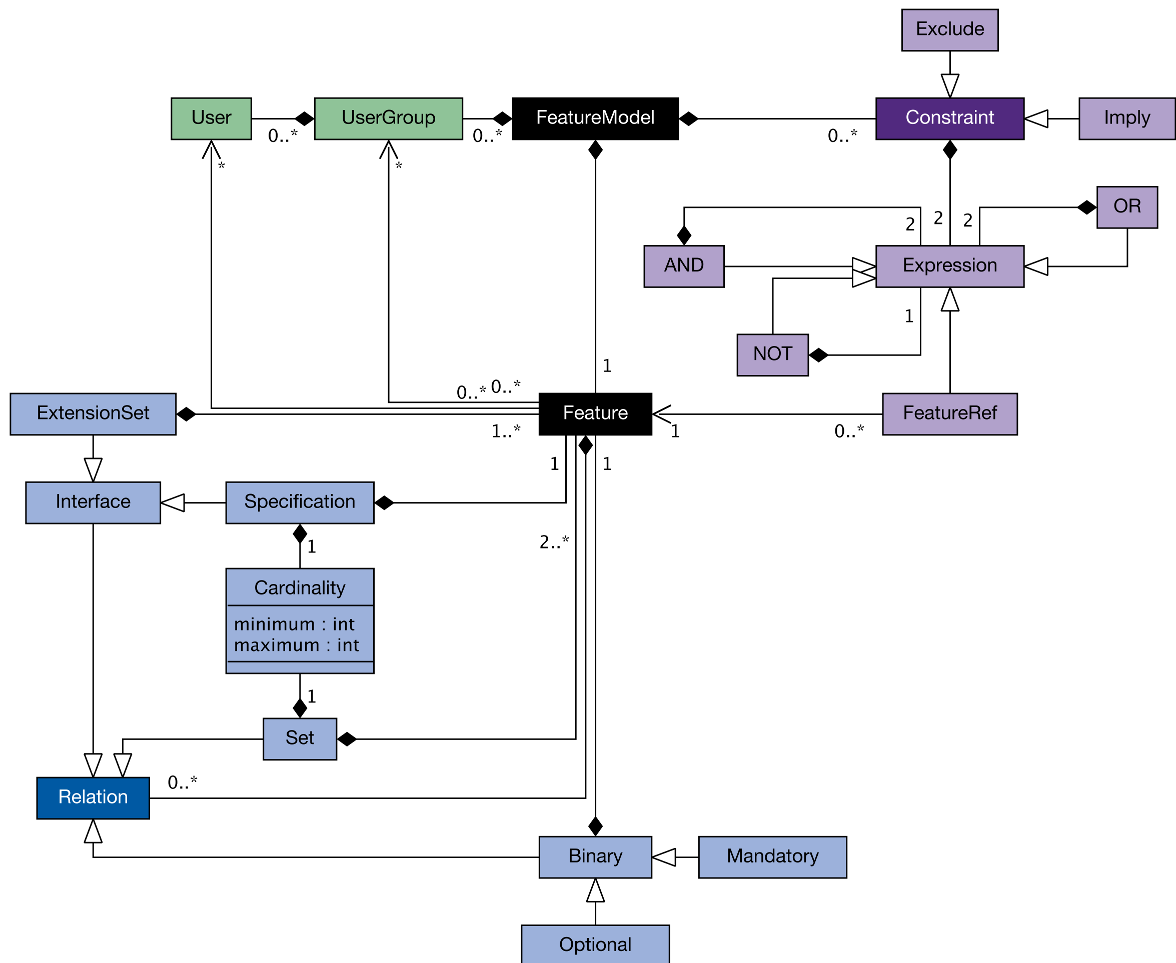


Figure 4.7 – Data structure abstraction for POSF feature models (UML class diagram)

Finally, the green upper-left part is used to implement the rights management and especially models the user management by the class `UserGroup` and its contained `User` class.

User Interface Conception. The modelling tooling needs some kind of user interface. Thus, a Graphical User Interface (GUI) is proposed to perform the modelling steps. As it is a prototype, it is sufficient to offer only basic functionalities and representation concepts by means of this GUI.

During the conceptual phase of the tooling, the GUI mockup shown in Figure 4.8 has been created. This mockup basically comprises a toolbar that allows basic loading and storing capabilities and a model area, separated into modelling of features and extensions, cross-tree constraints and user management. The actions that can be performed regarding the model are shown by means of a vertical toolbar and inside of drop-down menus. The model itself is visualised as a tree structure using the same controls filesystem browsers use.

Implementation Conception. Finally, the implementation has to be planned. Especially technological aspects need to be defined. The use of the Java programming language and Eclipse

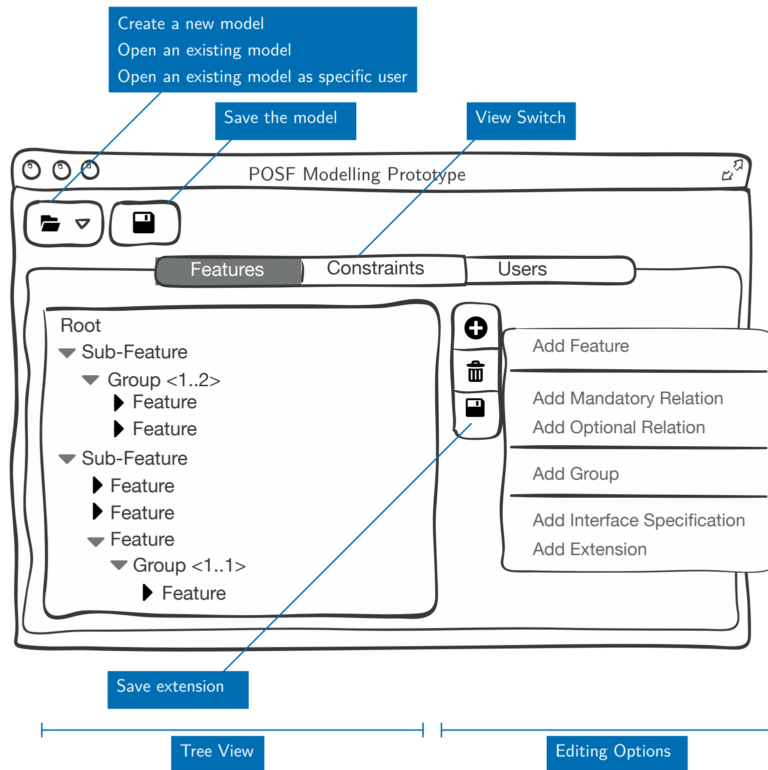


Figure 4.8 – GUI mockup for the prototype; representing the use cases of Figure 4.6

as IDE has been set at the beginning of the conception phase, because this potentially allows an easy integration of the implementation results into other academic projects of the Software Technology Group at the TU Dresden. This includes the use of the Eclipse Modeling Framework (EMF) as well. Additionally, the EMF Client Platform (ECP) can be used to generate a GUI based on the underpinned data structure. The support for a rights management has to be implemented by overriding the generated Java source code.

4.2.2 Implementation

The implementation of the prototype comprises the following steps:

1. Setting up the development environment to implement the prototype according to the conception
2. Modelling the data structure as Ecore model
3. Generating code from the Ecore model
4. Setting up the run configuration to initially implement a GUI
5. Modifying the Ecore model to realise all use cases by means of the used ECP
6. Overriding code to improve the GUI output
7. Manually implement the rights management

The IDE setup is part of the implementation as running the prototype requires a properly configured instance of the Eclipse IDE. Therefore, the Eclipse Modeling Framework has to be installed as well as the EMF Client Platform to support auto-generated GUIs.

The creation of an appropriate Ecore model is crucial to create a well-functioning prototype. Therefore, Figure 4.9 shows the graphical representation of the implemented Ecore model. In a first step, the Ecore model has been designed exactly the way Figure 4.7 intended. Some minor changes were necessary to perform all modelling steps properly using the given ECP GUI. For instance, adding cross-tree constraints is allowed in the form of child-elements of interfaces to include created constraints regarding an extension into an exportable sub-tree of the model. This is necessary due to a lack of multiple selectable items when exporting parts of the model from within the ECP application.

The code generation comprises the generation of the model code and the edit code. Hence, not all code generation processes offered by EMF are necessary to run the prototype using ECP.

To execute the prototype, a proper run configuration is required that explicitly invokes the ECP libraries and includes the needed features. Further information on how to set up the run configuration can be found in Appendix B, which is a user guide with detailed instructions on how to install and use the prototype.

The manually implemented code comprises appearance-related code and code that is necessary to implement the intended rights management. The prototypical rights management implementation shows how to obey the visibility rules by crossing out all model elements that are invisible to the currently selected user.

Figure 4.10 shows the final user interface of the prototype comprising the model in the left part of the screen and editing space on the right to modify element properties for each node in the left-hand tree.

Adding and removing elements can be performed via the elements' context menu within the model. Modifications such as user assignments, renaming and adding semantics information can be performed within the edit pane on the right. Further information on how the prototype looks and how to work with it can be found in Appendix B.

4.3 Summary

This chapter took up the results of the former chapters to extend feature models to support the open approach of POSFs. Therefore, the necessary steps were determined, which have to be performed to fulfill the in advance stated requirements for a sufficient POSF modelling strategy. These steps were performed successively in the Sections 4.1.1 until 4.1.4 to move towards a sufficient model.

Within these steps an interface notation and specification method has been proposed. Furthermore, different ways were presented to integrate semantic information to those interfaces as feature models only support modelling structural information. Finally, a possible rights management approach has been presented to allow modelling access restrictions for multiple contributors.

What is missing yet is a strategy to reduce the change impact level of this model. This is the goal of the next chapter. Starting with a change impact analysis first, it will conclude with possible rules guaranteeing a low change impact level.

To evaluate the new proposed model, a case study and evaluation will be performed in the chapter after the following. Therefore, a prototypical modelling tool has been implemented

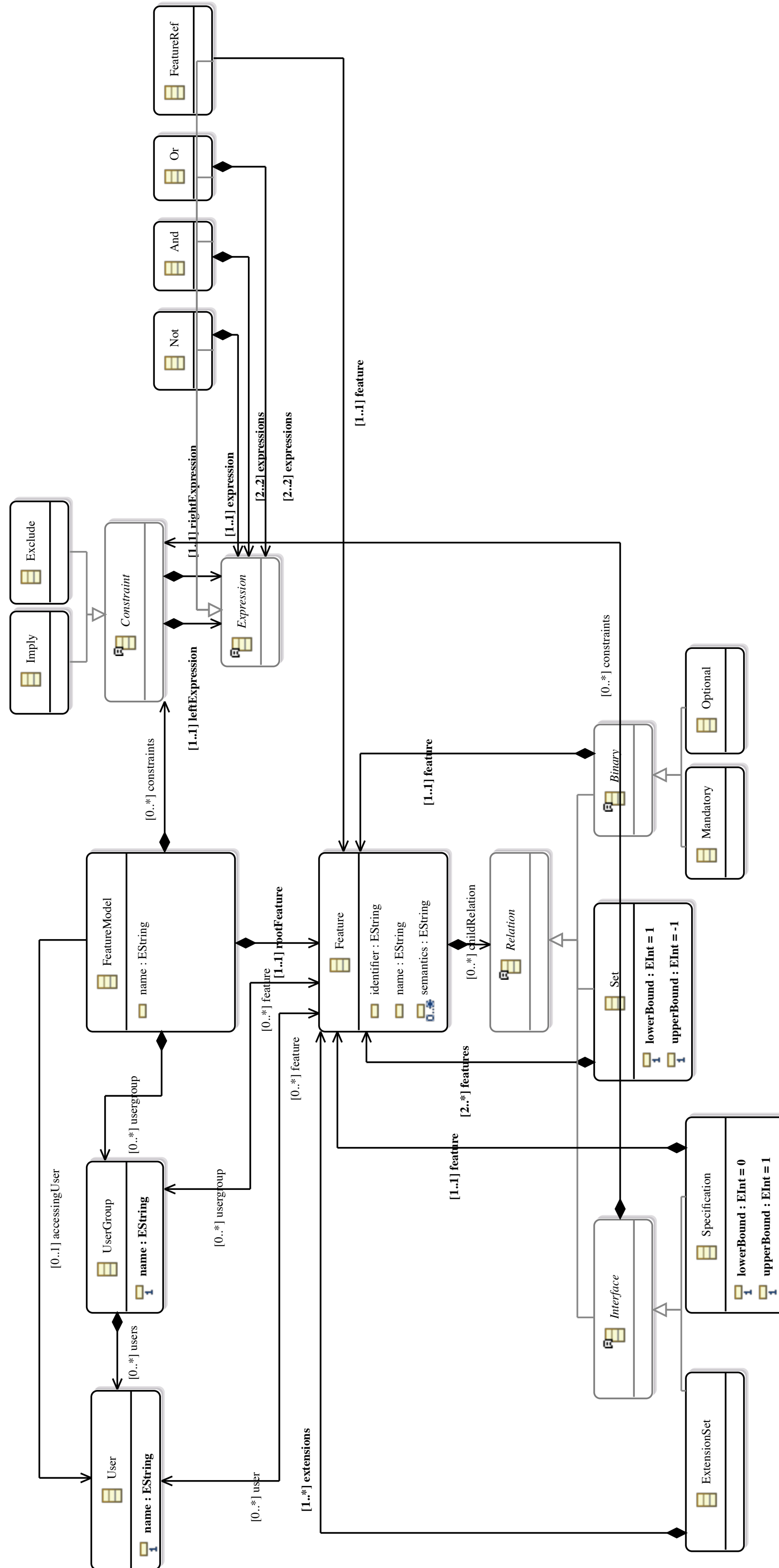


Figure 4.9 – Graphical representation of the underlying Ecore model based on the data structure of Figure 4.7

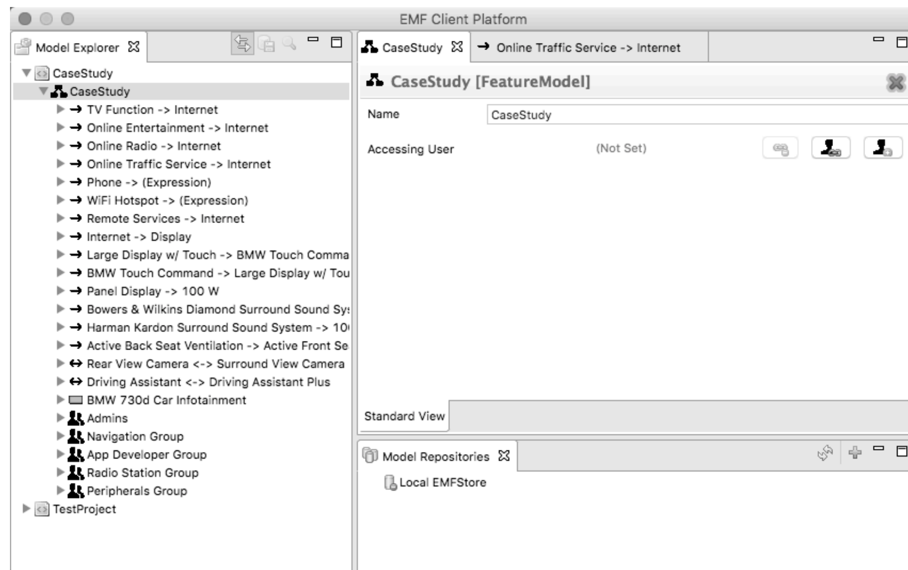


Figure 4.10 – Final GUI of the prototype

within Section 4.2 of this chapter. The conceptual and realisation phases were described there. A user guide on how to handle the tool can be found in the appendix of this thesis.

5 Analysis of the Change Impact

The former chapter proposed a new modelling method that supports variability modelling for Partly Open Software Families. This new model sufficies all stated requirements as proposed in Figure 2.7 of Chapter 2 but lacks one: the low change impact.

To tackle that issue, a change impact analysis is necessary to determine, which changes can be made to a model and with what potential impact. This is the task to fulfill within Section 5.1. A definition for the term *change impact analysis* is proposed in Definition 5.1. The results shall be used to derive rules that limit the permitted changes to reduce the change impact. This is the task of Section 5.2.

Definition 5.1: Change Impact Analysis

"Software change impact analysis, or impact analysis for short, estimates what will be affected in software and related documentation if a proposed software change is made."[BA96]

5.1 Change Impact Analysis

The purpose of a software change management and, thus, the actual impact analysis, is to derive answers to the following questions (based on [Chu03]):

- Who is affected by the changes?
- Where in the source code do changes need to be made?
- What else is affected by these changes?
- Why do we need to make all of these changes?
- When should all of the changes be made?

According to these questions, Figure 5.1 visualises the process of a change impact analysis in the software field. Within a first step, the changes that shall be made have to be determined (*change set*), an effect estimation based on this change set will be created (*estimated impact set*) and finally, the change will be implemented and the *actual impact set* can be derived from it.

The change impact analysis regarding the presented POSF-supported feature modelling is based on the first two steps of the above described analysis process. At first, the realisable changes will be identified and the potential change impact will be analysed afterwards. This helps to determine, which of the changes have potentially unwanted effects and where it is advisable to limit the permitted changes by modelling rules with the intention to reduce the change impact.

Regarding the above mentioned questions, the following questions can be derived from them as relevant for the afterwards following analysis:

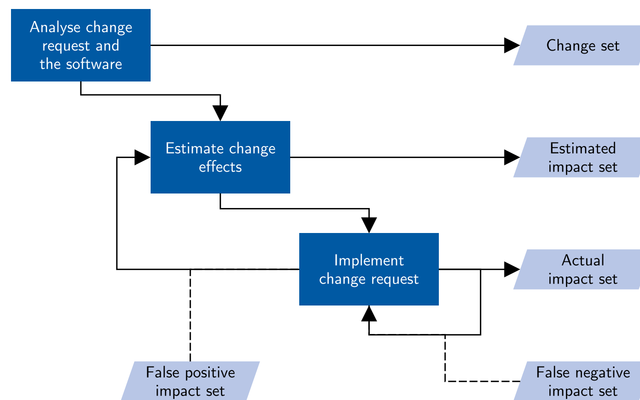


Figure 5.1 – Process visualisation of the change impact analysis, cf. [Li+13, p. 4]

- Does a change potentially affect other contributors supposing that external contributors mainly work with interfaces and extensions, while the main model (the model without considering the interfaces) is only modifiable by model maintainers (internal contributors)?
- What actual changes shall be performed?
- What is the potential effect of a change?

5.1.1 Possible Changes

The second of the above mentioned relevant questions (What actual changes shall be performed?) marks the first step of the analysis. All potential model changes have to be derived from the model. The systematic procedure comprises the following steps:

1. Identification of all model elements
2. What can be done to this element to change the model?
3. Are there distinguishable areas of the model that require a separate consideration?

What are the elements of a POSF model? A model comprises *features*, *relations* and *constraints*. Changes to the user management (how users are stored) are not relevant for this analysis. Thus, they will be left out here. Relations have to be separated in binary relations (mandatory and optional relation), set relations (groups) and interface relations (interface specification and concrete extension sets).

What changes can be done to these model elements? *Features* can be added and removed as well as changed, whereas “changed” means renaming or changing related underpinned source code. For a model-related change impact analysis, changing code does not have an effect on the model itself.

Relations can be changed in the following ways:

- A binary relation can be toggled from mandatory to optional and vice versa.

- A group relation can be changed by modifying the assigned upper and lower bound cardinality.
- A binary relation may be replaced by a group or vice versa.
- Changing interface relations is not relevant from this perspective except changes to the extension cardinality, which could lead to the invalidation of existing variants of a feature model. The validity of the model itself is not affected here. Other changes cannot be performed.

Constraints can be added or removed. Changing a constraint may be interpreted as an atomic constraint removal and addition.

What are model areas that require a separate consideration? It might be useful to analyse the "closed" and "open" aspect of a model separately. Thus, changes to the main model are analysed in a first step, while interface-related changes will be considers afterwards.

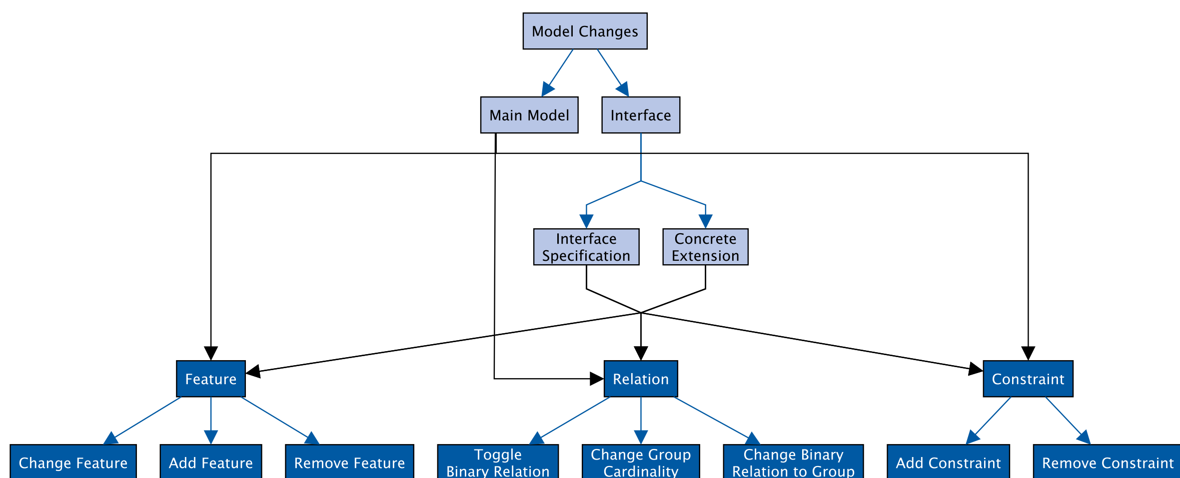


Figure 5.2 – Graph of potential model changes

Figure 5.2 visualises the mentioned changes as a diagram. The lighter-coloured top part of the diagram shows the considerable areas of the model. The bottom part of the model shows the model elements and their performable changes to analyse.

5.1.2 Identification of the Influence Scope

The next step of the analysis is to identify the potential impact of each beforehand mentioned changes. A special focus is to identify whether other model contributors may be affected or not. The assumption, therefore, is that there are model maintainers that access the "closed" part of the model and extension contributors that limit their work to accessing interfaces respectively the "open" part of the model.

Additionally, if contributors are affected by a change, the rights management allows an identification of which users are affected. This can be done by tracing the path of an affected feature up to the root feature. According to the user-feature mapping, all users can be identified that have modification rights to a feature. These users are potentially affected by a change. This

is an important advantage of POSF modelling compared to other modelling strategies without any user and rights management.

Changes Regarding the Main Model

The main model comprises the whole feature model except the interfaces (specifications and extension sets).

Feature Modification – Change a Feature Renaming a feature is a local change that only affects the feature itself. Performing this within the root model has no effect on other features and, thus, on other contributors.

Feature Modification – Add a Feature Adding a feature potentially increases the variant space. The step of adding a feature comprises the addition of a new relation, too. However, none of these relations could lead to an actual reduction of the variant space as no former allowed configuration could be invalidated. There is only one exception to that: the special case of a mandatory relation. In this case, each former allowed configuration that contains the parent feature of the added one has to contain the new feature, too. Otherwise, this variant would be invalidated. Nevertheless, this has no impact on the cardinality of the variant space, further parts of the model and other contributors.

Feature Modification – Remove a Feature Removing a feature reduces the variant space if there were valid variants that contained the removed feature. Removing a feature removes the relation to its parent as well and leads to the removal of the by the removed feature defined sub-tree of the model. Hence, this change is potentially invasive. If there were implication cross-tree constraints with the removed feature as consequence, the removal leads to an invalidation of the model. The premise part of the implication indicates the affected area of the model. This may affect external contributors if the sub-tree contained an interface, or the mentioned implication contained a feature of an extension in its premise. Other constraints could have a similar effect.

Relation Modification – Toggle a Binary Relation Toggling a binary relation from mandatory to optional has no effect on other parts of the model as the possible variant space is not reduced by this. The feature may still be selected. On the other hand, this feature is not required anymore, which may lead to an increased variant space. Other contributors are not affected at all. If a relation is turned from optional to mandatory, the variant space may be reduced. In consequence, even other contributors may be affected here in case a cross-tree constraint leads to invalidity of an interface if the – now mandatory – feature is selected.

Relation Modification – Change the Group Cardinality Changing the upper or lower bound cardinality of a group relation may lead to a change of variant space's size. If the lower bound is increased or the upper bound decreased, the variant space will potentially be reduced as a more limited amount of sub-features has to be selected to derive a valid variant. If the lower bound is decreased or the upper bound increased, this may lead to a larger variant space as no former valid variant will become invalid. If both values are increased or decreased, the actual effect cannot be estimated in advance. In each case of variant space reduction other contributors are affected whenever an interface requires features that cannot be selected from now on. Consider an XOR group with two features A and B. If there is a feature X that is mandatory to each

variant and that implies feature A, there is no chance to get feature B as part of any valid configuration. If this unreachable feature B is required by an interface or contains an interface, the contributors of those interfaces are affected by invalidation in form of unreachability of their extensions.

Relation Modification – Change a Binary Relation Into a Group Changing binary to group relations and vice versa may change the set of selectable features of the related parent feature. If this reduces the variant space, the case is identical to the change of the group cardinality. In any case of variant space increase or an unchanged variant space, this change would not have any effect on other contributors.

Constraint Modification – Add a New Constraint Constraints reduce the variant space. They could lead to model invalidation, too. The above mentioned example of features A, B and X with added implication $X \rightarrow A$ illustrates this. This is the case for exclusions as well. This may affect other contributors the same way as described in the change of a group cardinality. Additionally, constraints may exist across the main model, interface specifications and concrete extensions. These constraints affect other contributors directly.

Constraint Modification – Remove a Constraint The removal of a constraint increases the variant space and, thus, does not affect other contributors or other parts of the model.

Changes Regarding an Interface Specification

The following part of the analysis is limited to interface specifications. Changes of the interface cardinality potentially lead to the invalidation of existing variants if the cardinality is more restrictive.

Feature Modification – Change a Feature Changes to the interface specification always affect all contributors who develop extensions for this interface. Thus, even renaming a feature in this specification leads to the invalidation of existing concrete extensions. However, the regarding features of the concrete extensions simply need to adapt the renaming to regain validity. Thus, the change is minimal. Constraints that contain a feature reference to that changed feature need to adapt the new name as well. Else, there would be an unsatisfiable constraint, which could lead to a variant space reduction.

Feature Modification – Add a Feature Adding features to an interface specification leads to a more specific interface description. Hence, concrete extensions are affected here, because they need to adapt the changes to stay valid. Thus, the developers of these extensions are affected by this change.

Feature Modification – Remove a Feature Removing a feature out of an interface specification leads to a less detailed specification. This does not invalidate existing concrete extensions. Thus, there is no direct negative impact on other contributors.

Relation Modification – Toggle a Binary Relation, Change the Group Cardinality, Change a Binary Relation Into a Group Changing a relation inside of an interface specification has to be carried over by existing concrete extensions. These changes affect the other extension developers directly in a similar way as the same changes to the main model do.

Constraint Modification – Add a New Constraint Adding a constraint to an interface specification is similar to that for main models if these constraints refer to features across the whole tree (i. e. outside of the interface). Additionally, the constraint has to be carried over by the existing concrete extensions to keep their validity. Hence, all contributors regarding that interface are affected again.

Constraint Modification – Remove a Constraint Removing a constraint does not reduce the variant space and is not required to be carried over to the existing concrete extensions. Thus, this has no affect to other contributors of the model in general.

Changes Regarding a Concrete Extension

The following last part of the scope analysis is limited to the extension sets of interfaces. Changes to the extension set (adding and removing extensions) are not considered here as these changes are no real model contributions but rather relevant for configuration purposes. The extension set is always variable.

Feature Modification – Change a Feature Renaming a feature of a concrete extension only has an effect if the feature is part of the interface specification. If so, this change would invalidate the extension. Else, the change has no effect.

Feature Modification – Add a Feature Adding a feature is explicitly allowed and has no effect on other contributors or on the model.

Feature Modification – Remove a Feature Removing a feature is as invasive as the removal of features of the main model. The consequences are equal. Other contributors are potentially affected if there is an implication constraint that becomes unsatisfiable by that removal. As there is no restriction to nested interfaces, other contributors regarding interfaces inside of the extension are affected here. One special case: If the removed feature is part of the interface specification, the extension becomes invalid immediately.

Relation Modification – Toggle a Binary Relation, Change the Group Cardinality, Change a Binary Relation Into a Group Changing relations leads to invalidity if the relation is part of the interface specification. In every other case, the effects are the same as relation changes inside of the main model. Thus, if the variant space is reduced, other contributors may be affected (through cross-tree constraints or nested interfaces). Else, there is no effect on other contributors of the model.

Constraint Modification – Add a New Constraint Adding constraints with feature references that only affect features of the extension does not have any effect on other contributors or the model as a whole. But they may reduce the variant space added by the extension itself.

If a constraint is added that encompasses feature references across the whole feature model (including the main model), the whole model and all contributors are potentially affected.

Constraint Modification – Remove a Constraint The removal of a constraint cannot reduce the variant space and, thus, should have no effect on other contributors.

5.1.3 Potential Overlap with Other Contributors

All changes to a feature model do have an impact on the model. In most cases, these changes lead to an increase or a reduction of the variant space, which is the set of all valid variants of the feature model. An increase is no problem in most cases. A reduction is potentially “dangerous”, because this could lead to an invalidation of parts of the model or even the entire model. In consequence, desired variants cannot be validly derived.

However, those changes are immanent for feature models and it is possible to check the model for undesired effects. A concrete problem arises whenever other model contributors are affected by changes. For the presented POSF models this is the case whenever the variant space is potentially reduced by a change, and the reduction comprises interfaces or a subset of the by means of an extension derived variant space.

According to the analysis of Section 5.1.2, the following changes have to be considered as potentially affecting other contributors:

- Changes regarding the main model
  - Feature Modification – Remove a Feature
  - Relation Modification – Toggle a Binary Relation
  - Relation Modification – Change the Group Cardinality
  - Relation Modification – Change a Binary Relation Into a Group
  - Constraint Modification – Add a New Constraint
- Changes regarding interface specifications
  - Feature Modification – Change a Feature
  - Feature Modification – Add a Feature
  - Relation Modification – Toggle a Binary Relation, Change the Group Cardinality, Change a Binary Relation Into a Group
  - Constraint Modification – Add a New Constraint
- Changes regarding concrete extensions
  - Feature Modification – Remove a Feature
  - Relation Modification – Toggle a Binary Relation, Change the Group Cardinality, Change a Binary Relation Into a Group
  - Constraint Modification – Add a New Constraint

5.2 Minimisation of Unwanted Impact

Section 5.1.3 shows that nearly every possible model change could potentially affect other model contributors. The introduction of a modelling rule similar to the following is infeasible, although this rule seems natural.

> *Every change is permitted as long as no other model contributor is affected by a variant space reduction.*

Especially changes to the main model would be denied in many cases, for instance if a feature is intended to be removed by a maintainer. That is why unwanted impact cannot be ruled out entirely by means of the introduction of a modelling rule. However, a rule seems to be more feasible when focussing on the risky changes that do not allow a reliable impact estimation. That is of vital concernment.

This is the case whenever external extensions influence the variant space implied by the main model. This is caused by cross-tree constraints that are introduced by concrete interfaces and that encompass feature references over the whole model. To avoid those critical changes, the following rule should be introduced:

> *Concrete extensions are not allowed to introduce cross-tree constraints that contain feature references to features outside of the extension. If there is a constraint between features of extensions and features of the main model necessary, this has to be introduced by the interface specification.*

The introduction of this modelling rule helps to avoid changes that are entirely unpredictable. It requires cross-tree constraints to remain inside of the main model and the interface specifications or remain inside of one concrete extension.

On the other hand, this rule may be too restrictive as constraints between an extension and the main model are required in. Thüm et al. [TBK09] presented a way to classify feature model changes in one of the following categories:

- Refactoring (no variant is added nor removed from the variant space)
- Generalisation (variants are added but not removed from the variant space)
- Specialisation (no variants are added but removed from the variant space)
- Arbitrary edit (variants are added and removed from the variant space)

Changes that belong to the first two categories can be performed without affecting any contributor or invalidating variants. Changes that belong to one of the other two categories lead to an invasive change to the variant space. In consequence, contributors may be affected. In case such a change has to be performed, the affected contributors should be notified in advance. To do this, the model's rights management supports the identification of the affected users. This method is proposed as a more liberal alternative to avoiding cross-tree constraints that refer to extensions and the main model as stated above.

5.3 Summary

This chapter performed a systematic change impact analysis by identifying the possible changes that have to be considered, determined the impact scope of each change and especially determined whether a change may affect other model contributors and which of them. Finally, the potentially critical changes were named and a modelling rule is introduced that leads to an isolation of cross-tree constraints to an uncritical model area. As an alternative tradeoff, findings from Thüm et al. were used to identify uncritical model changes that can be performed. In any critical case, the affected contributors have to be identified and notified in advance of the actual change.

The next chapter will take up the proposed POSF model and especially the developed prototype and will perform an evaluation using a case study based on a car infotainment system that is built on the 2015 BMW 7 series as an example.

6 Evaluation and Case Study

This chapter evaluates the POSF modelling concept using a case study based on the structure of a car infotainment system. To do this, the actual case study is developed in a first phase. The second phase comprises the accomplishment of all six use cases as stated in Figure 4.6. Each of the performed use cases is discussed afterwards based on the defined modelling requirements.

6.1 Conception

Before an evaluation can be performed, a case study has to be built. Therefore, a data basis is required to construct a potential software family as a POSF. One motivation to introduce the POSF concept is the idea of a customisable car software system. Thus, a car (and especially its software related parts) can be configured as usual when buying a new car at the vendor's side which follows the product line approach. But additionally, the customer should have the opportunity to add, exchange and remove certain functionalities of the software-related parts afterwards on the customer's side. For instance, a list of online radio stations could be extended this way or navigation maps could be exchanged, updated or extended. The installation of additional software to support proprietary smartphone protocols could be possible, too. These functionalities are developed by external developers similar to apps for smartphones. This open approach is known from ecosystems. Finally, such a car should be developed as a mixture of a product line and an ecosystem, leading to a customisable platform with interfaces to plug extensions in.

The car infotainment system has been used as a running example in former chapters. The case study shall be built on that idea but comprise a realistic set of features and interfaces. Thus, the configurable part of a real car's infotainment is used as a foundation. Furthermore, some features need to be added that are necessary for an infotainment system, but which are not visible to customers during the configuration process. Finally, to introduce openness, some points of the model are chosen to introduce interfaces, which support the required openness. Appendix C shows the composition process of features and functionalities to a tree-structured model with indication of the source each of the included features is taken from.

For the basis of a real car, the 2015 BMW 7 series is chosen as it comprises a variety of different infotainment system-related functionalities and features. [BMW15a; BMW15b; BMW14]

The created model comprises twelve top-level features as children of the root feature. These are shown in Figure 6.1. These features can be seen as categories of the features, which are modelled as shown in the Figures 6.2 until 6.13. The following numbers can be derived from this model:

- There is 1 root feature (*BMW 730d Car Infotainment*).
- There are 12 top-level features.
- There are 90 features and 9 interfaces in total.

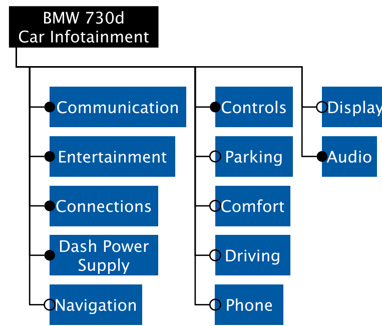


Figure 6.1 – Case study structure – top-level structure

- *Communication* (Figure 6.2) describes multiple mobile (online) services and contains
  - 9 features
  - 1 interface (support for additional web browsers)
- *Entertainment* (Figure 6.3) describes multimedia features such as radio and tv as well as apps such as a Facebook and Twitter app. The feature contains
  - 12 features
  - 2 interfaces (allow adding online radio stations and online apps)
- *Connections* (Figure 6.4) describes communication protocols such as WiFi and Bluetooth and contains
  - 10 features
  - 1 interface (support for new communication protocols)
- *Dash Power Supply* (Figure 6.5) describes electric power settings and contains
  - 2 features
- *Navigation* (Figure 6.6) describes all components of the navigation system and contains
  - 11 features
  - 2 interfaces (support to add external maps and online real-time traffic services)
- *Controls* (Figure 6.7) describes the different controlling elements to interact with the infotainment system. This feature contains
  - 5 features
  - 1 interface (allow adding additional controlling gestures)
- *Parking* (Figure 6.8) contains the parking assistant functionalities and, therefore, contains
  - 5 features
- *Comfort* (Figure 6.9) describes in-car climate comfort features and contains
  - 5 features
- *Driving* (Figure 6.10) describes features related to the driving comfort and, therefore, contains
  - 4 features
- *Phone* (Figure 6.11) describes the phone integration features and contains
  - 6 features
  - 2 interfaces (support for specific bluetooth and Universal Serial Bus (USB)-connected phones)
- *Display* (Figure 6.12) describes the GUI related features and contains
  - 5 features

- *Audio* (Figure 6.13) contains the available speaker systems and comprises three options as
  - 3 features

Table 6.1 – Case study – cross-tree constraints

Type	Constraint
Implication	$TVFunction \rightarrow Internet$ $OnlineEntertainment \rightarrow Internet$ $OnlineRadio \rightarrow Internet$ $OnlineTrafficService \rightarrow Internet$ $Phone \rightarrow Bluetooth \wedge USB$ $WiFiHotspot \rightarrow WiFi \wedge Internet$ $RemoteServices \rightarrow Internet$ $Internet \rightarrow Display$ $LargeDisplayWithTouch \rightarrow BMWTouchCommand$ $BMWTouchCommand \rightarrow LargeDisplayWithTouch$ $PlanelDisplay \rightarrow 100W$ $Bower\&WilkinsDiamondSurroundSoundSystem \rightarrow 100W$ $HarmanKardonSurroundSoundSystem \rightarrow 100W$ $ActiveBackSeatVentilation \rightarrow ActiveFrontSeatVentilation$
Exclusion	$RearViewCamera \leftrightarrow SurroundViewCamera$ $DrivingAssistant \leftrightarrow DrivingAssistantPlus$

In addition to the tree-structure of the integrated features, 16 cross-tree constraints are given (cf. Table 6.1). For instance, if the *WiFi Hotspot* feature is selected, the features *WiFi* and *Internet* have to be selected too. Furthermore, there could either be the *Driving Assistant* or the *Driving Assistant Plus* selected but not both. Selection in this context means that a feature will be part of a concrete variant. Analogously, if a feature is not selected, it is not part of the variant.

Table 6.2 – Case study – users and user groups

Admins Feature(s): BMW 730d Car Infotainment	Navigation Group	App Developer Group Feature(s): Online App, Browser, Gestures	Radio Station Group Feature(s): Station	Peripherals Group Feature(s): Peripherals Protocol, Bluetooth Phone, USB Phone
Administrator	Map Contributor Feature(s): Additional Map	One External App Developer	Station Distributor	Some Peripherals Supplier
	Online Traffic Service Contributor Feature(s): Online Traffic Service	Second External App Developer		Phone Manufacturer

To evaluate the rights management, some users and user groups are integrated into the model. Table 6.2 gives an overview of them. This case study comprises five user groups.

Admins Each user of this group shall be able to see the entire model. Thus, there is a feature assignment of the root feature to this group.

Navigation Group This group comprises all users that contribute to the navigation system. The user *Map Contributor* may see the *Additional Map* interface to add extensions. The user *Online Traffic Service Contributor* is permitted to see the *Online Traffic Service* interface to develop a new service extension here.

App Developer Group This group is a set of all contributors that shall be allowed to extend the functionality of online apps. Therefore, all users of this group are permitted to see the *Online App*, *Browser* and *Gestures* interfaces to develop extensions there.

Radio Station Group Online radio stations can be included as extensions to the car infotainment system. The Radio Station Group represents the set of station contributors. They all are permitted to see the *Station* interface to distribute extensions to that.

Peripherals Group Hardware manufacturers (especially smartphone manufacturers) may produce peripherals that can interact with the car. To implement device-specific features, these manufacturers are allowed to access the model as a user of the Peripherals Group. Thus, those contributors are permitted to access the *Peripherals Protocol*, *Bluetooth Phone* and *USB Phone* interfaces.

Based on this case study construction, the case study can be performed by doing the actual modelling using the prototypically developed tool and by exemplarily modelling an extension to that car infotainment system.

6.2 Perform the Case Study by Means of the Developed Tool

Conducting of the above presented case study comprises the realisation of the model using the prototypical tool including the feature structure, cross-tree constraints, user groups and users. Additionally, an extension shall be modelled exemplarily.

All of these steps are structured using the prior defined use cases of the prototype (cf. Figure 4.6). After this conduct, Section 6.3 checks whether the stated POSF requirements are satisfied or not. The case study performance comprises the following tasks:

1. Model the case study construction.
   a) The creation of a new model project within the prototype.
   b) The feature structure realisation given in Figure 6.2 until Figure 6.13.
   c) The modelling of cross-tree constraints based on Table 6.1.
2. The interface modelling based on the structure given in Figure 6.2 until Figure 6.13.
3. The creation of user groups, users and feature assignments based on Table 6.2.
4. The modelling process of a concrete extension using one of the modelled interfaces.

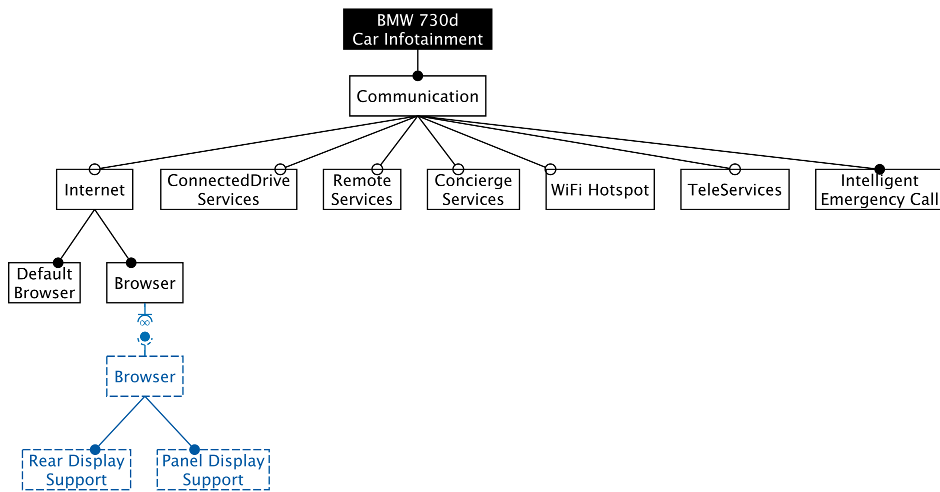


Figure 6.2 – Case study – derived feature model part: Communication

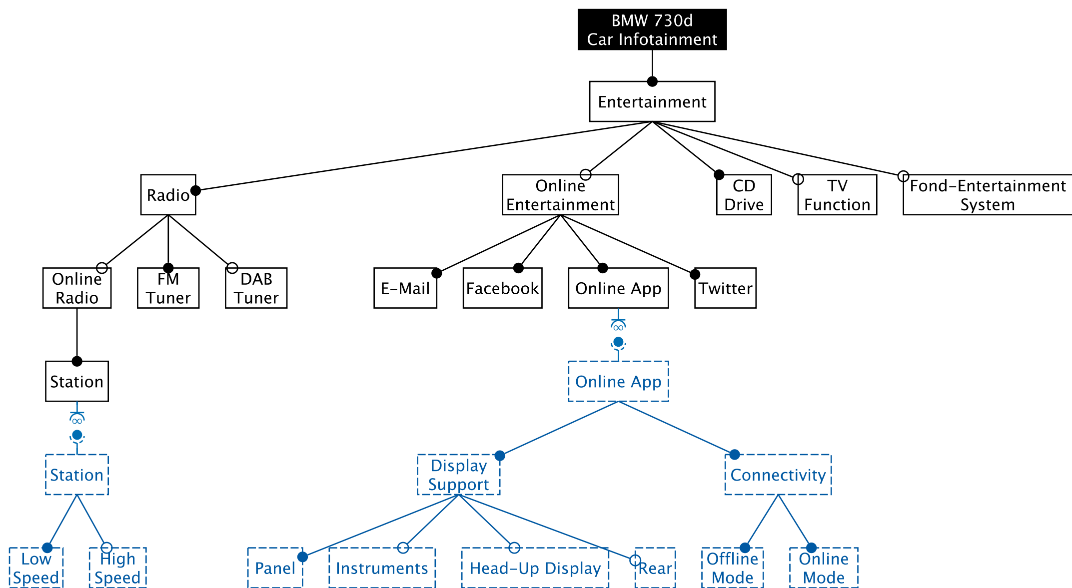


Figure 6.3 – Case study – derived feature model part: Entertainment

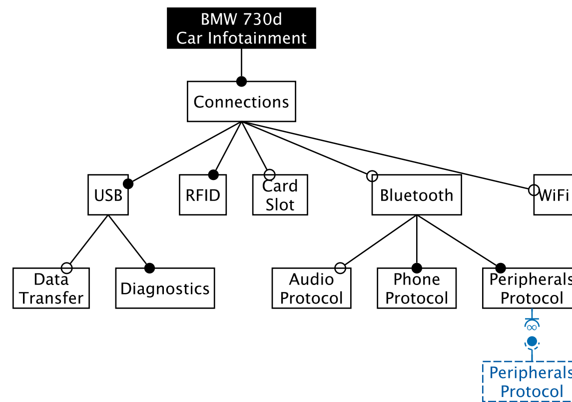


Figure 6.4 – Case study – derived feature model part: Connections

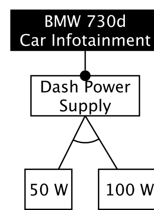


Figure 6.5 – Case study – derived feature model part: Dash Power Supply

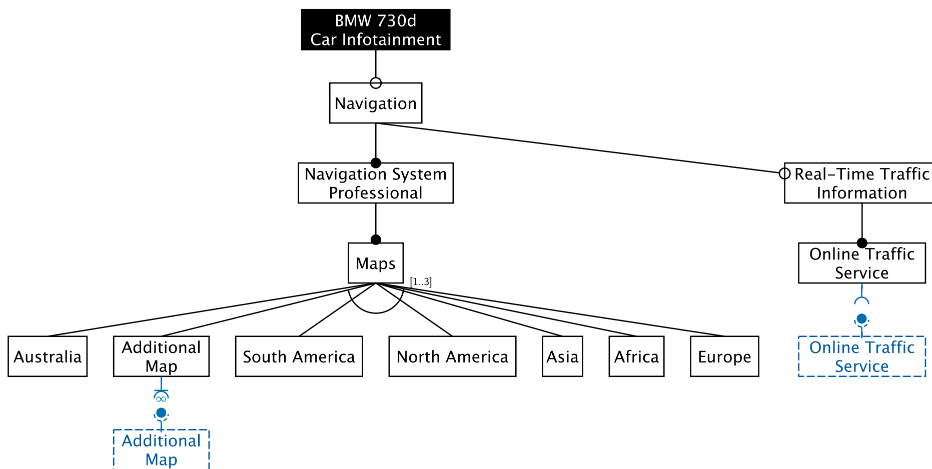


Figure 6.6 – Case study – derived feature model part: Navigation

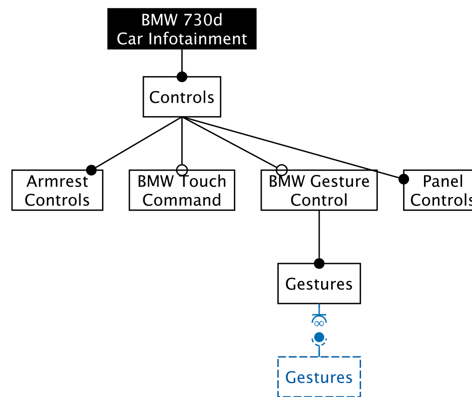


Figure 6.7 – Case study – derived feature model part: Controls

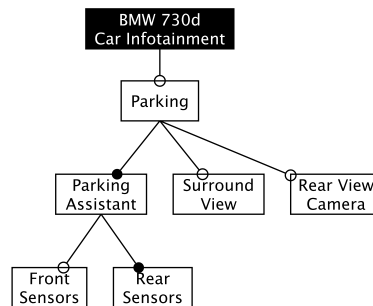


Figure 6.8 – Case study – derived feature model part: Parking

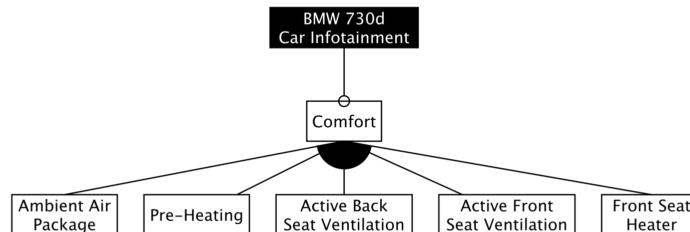


Figure 6.9 – Case study – derived feature model part: Comfort

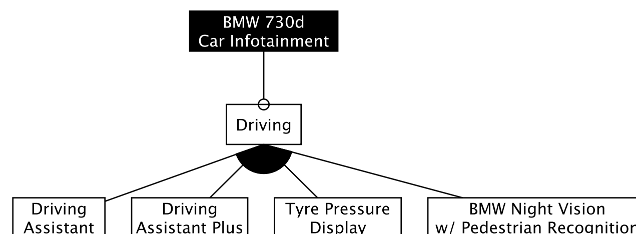


Figure 6.10 – Case study – derived feature model part: Driving

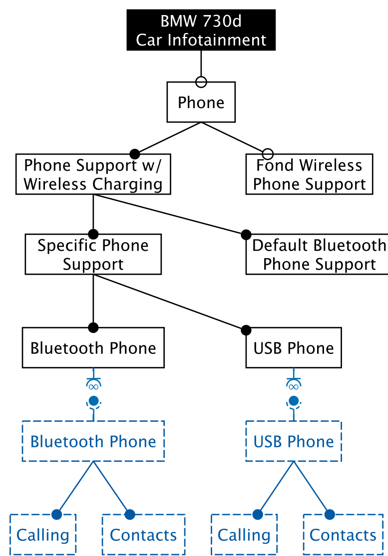


Figure 6.11 – Case study – derived feature model part: Phone

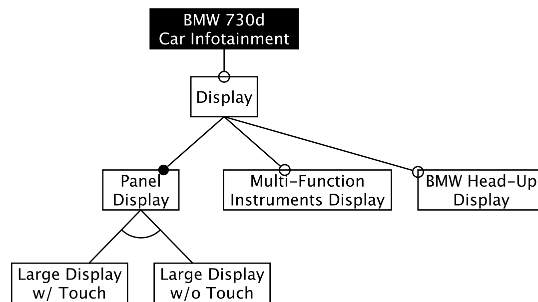


Figure 6.12 – Case study – derived feature model part: Display

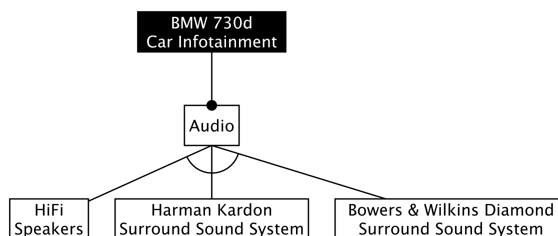


Figure 6.13 – Case study – derived feature model part: Audio

6.2.1 Model the Case Study Construction (Open, Store and Edit Model)

The first step of performing the case study comprises the creation of a new modelling project, modelling the feature structure as well as cross-tree constraints. To perform these steps using the prototypical tooling, the first part of Appendix B have to be followed.

The creation of a new modelling project within the prototype comprises the creation of a project root element of type *Feature Model.* The child-elements of this project root are the actual feature model's root element called *BMW 730d Car Infotainment*, the cross-tree constraints based on Table 6.1 as well as the user groups as described below in Section 6.2.3.

Modelling the feature structure is based on the models shown in Figure 6.2 to Figure 6.13, except modelling the interfaces. The interface modelling is done in the following Section 6.2.2.

6.2.2 Model the Interfaces (Edit Model)

Modelling the interfaces comprises the inclusion of all interface specifications given in Figure 6.2 until 6.13. This includes the following nine interfaces and their structural specifics:

- *Browser* (which has to support a panel and rear display for the console and fond (rear seat) entertainment system)
- *Station* (to support online radio stations with a low and high speed connection audio stream)
- *Online App* (which supports multiple displays inside the car and supports an online and offline mode depending on the online service availability)
- *Peripherals Protocol* (to support proprietary hardware communication protocols)
- *Additional Map* (for the navigation system)
- *Online Traffic Service* (to get traffic information inside the navigation system)
- *Gestures* (to support additional gestures for the BMW Gesture Control)
- *Bluetooth Phone* and
- *USB Phone* (to support specific phone hardware at least for calling and with contact management capabilities)

This modelling process leads to adding an *Interface Specification* relation as child element to the parent interface feature. The cardinality has to be set to *[0..*]* for each of the interfaces; except the interface *Online Traffic Service* has to be set to *[1..1]* as there is one extension required if the *Real-Time Traffic Information* feature is selected inside a concrete variant of the infotainment system.

Additionally to the structural specification of these interfaces, semantics information can be added, too. As there is no one-fits-all solution for the semantics description, a feasible procedure for describing semantics has do be determined in a real world software project. Within this case study, a brief SMART-oriented semantics description shall be proposed for the online radio station:

Station The station extension comprises one entry for the list of online radio stations. This station has to display a station name and a station logo inside of that list. Additional programme-related textual information is allowed but not required to be displayed, too.

Low Speed The low speed option contains a weblink to the radio station audio steam. This audio stream has to be available in MP3 format and with a bitrate of 64 kbps.

High Speed The high speed option contains a weblink to the radio station audio steam. This audio stream has to be available in MP3 format and with a bitrate of 128 kbps.

6.2.3 Create Users and Assign Features (Edit Model and Open Model as User)

Users are organised inside of user groups. Therefore, the user groups are modelled as child nodes of the project root element. Based on Table 6.2, the following groups are created: *Admins*, *Navigation Group*, *App Developer Group*, *Radio Station Group*, *Peripherals Group*. According to the mentioned table, users are created inside these groups.

Each user, except the *Administrator* (who shall be able to access the whole model), shall exclusively be able to develop concrete extensions for each interface. Therefore, each user or user group gets the appropriate feature assignment for its specific purpose. Hence, the *Map Contributor* gets access permissions to the *Additional Map* interface to add a new map extension to the navigation system.

In this case study, a concrete extension is created using the *Station* interface. This should be done from the *Station Distributor's* perspective. That user is part of the *Radio Station Group*, which has access to the *Station* interface only.

6.2.4 Model a Concrete Extension (Edit Model, Store and Import Extension)

To evaluate the extension modelling as a crucial part of POSFs, an exemplary concrete extension is created. Therefore, the extension adds an online radio station to the system. The interface specifies that a station has to comprise two audio streams in different audio qualities (for low speed connections and high speed connections).

To model this extension, the modelling user has to have access to the *Station* interface. In this case study, the user *Station Distributor* shall provide that extension. As this user is member of the *Radio Station Group*, he has the required permission. He is permitted to see the interface, its specification, the path to the model's root feature and he has the permission to model a concrete extension to the *Station* interface. To switch the prototype perspective to the desired user's perspective, the user has to be assigned to the project root element. After closing and reopening the modelling project, the perspective switches to the *Station Distributor's* view.

Figure 6.14 demonstrates the visible part of the model (except the interface specification) and shows an exemplary concrete extension called *ExampleFM* (blue highlighted features), which comprises – as required by means of the interface – a mandatory *Low Speed* option and an optional *High Speed* option. The constraint

$$OnlineRadio \rightarrow Internet$$

is visible too because of the visible feature *Online Radio*.

To model the extension using the prototype, an *Extension Set* element has to be created as an interface child and next to the *Interface Specification* element. This extension set includes the concrete extension with its extension root feature *ExampleFM* and the structure of the blue highlighted part of Figure 6.14. Figure 6.15 shows the concrete extension inside the navigation view of the modelling tool. This extension may be exported to an **.xmi* file using the partly model export function at the extension's root feature as shown in Section B.4 and imported into another model's Station extension set using the import functionality as described in Section B.6.

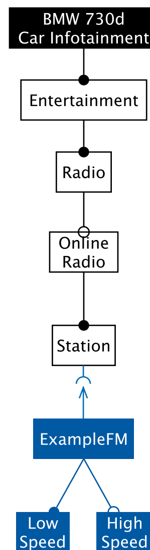


Figure 6.14 – Case study – concrete extension to the Station interface

6.3 Results and Discussion

The result of Section 6.2 is the model realisation of the car infotainment system as constructed in Section 6.1 using the prototype developed in the second part of Chapter 4. Additionally, a concrete extension has been integrated as a specific user in Section 6.2.4.

The evaluation of this case study comprises a requirements check to determine, whether the POSF requirements as stated in Section 2.2.1 are satisfied or not.

Modelling closed and open variability at the same time. The "closed" variability modelling describes the modelling process similar to SPLs, which comprises the definition of a fixed set of features and structuring them inside of a feature model. In this case study, Section 6.2.1 describes that part.

Modelling "open" variability describes the modelling process of interfaces and their specification that supports further model extension afterwards (referred to as *open* modelling). Section 6.2.2 describes the interface modelling phase of this case study.

As both of these processes are performed inside one single model based on the principles of Chapter 4, this proofs that it is possible to model closed and open variability at the same time. Hence, this requirement is satisfied.

Specification of extension interfaces. In conjunction with the open variability modelling of the above mentioned requirement, the specification of interfaces is already shown as part of this process. Hence, the specification of extension interfaces is possible (cf. Section 6.2.2) and this requirement is considered as fulfilled.

Allow decentralised extension development. It shall be possible to develop extensions externally, thus, in a decentralised way. This requirement contains two elements: The capability to model an extension and the portability of this modelling part to perform this externally and integrate the results into another model afterwards.

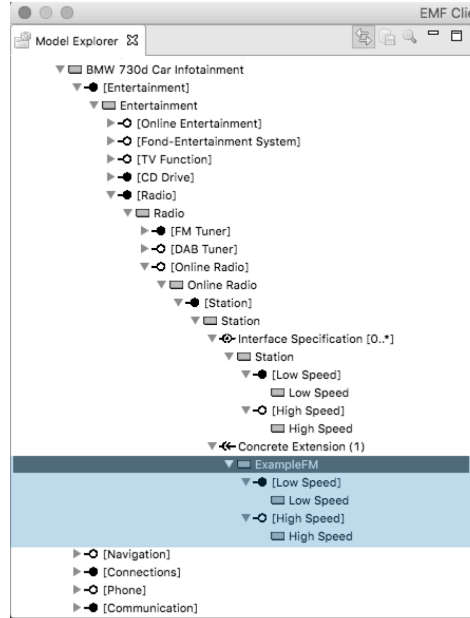


Figure 6.15 – Modelling a concrete online radio station extension using the prototype

Section 6.2.4 demonstrates both aspects using the developed prototype. Therefore, a concrete extension (an online radio station) is modelled, exported and re-imported into the model. The modelling process includes adding a concrete *Extension Set* element next to the *Interface Specification*, which allows the addition of concrete features or extensions to the model. The modelled extension can be exported, the project closed, re-opened and finally, the extension can be re-imported to the concrete set of extensions. Hence, the modelling procedure can be performed externally and following a decentralised approach. This requirement is fulfilled.

Allow multiple contributors to modify the model. The multi-user support of the model requires the introduction of multiple users. This is done in Section 6.2.3 of this case study. Accessing the model as a specific user is performed during the extension modelling of Section 6.2.4 as the user *Station Distributor* contributes to the model. Because this can be performed externally, the contribution to one model by multiple users is shown.

One aspect that is not shown during the case study performance is the parallel modification of the model as the prototype does not support simultaneous access yet. However, this is not explicitly demanded by the requirement but could lead to synchronisation problems – similar to the problem of simultaneous access to text documents or spreadsheets using an online service such as Google Docs or Microsoft Office Online.

Nevertheless, the model modification through multiple contributors is supported by the model as shown above, which leads to the satisfaction of that requirement.

Rights management for multiple contributors. The requirement for a rights management has the intention to integrate a controlling unit to permit or deny access to certain elements or parts of the model. This access includes the the permission to see or to modify a part of the model.

The POSF modelling as described in Chapter 4 implements this rights management by introducing users (and user groups) that can be assigned to features. This feature assignment leads to certain visibility and modification rules as described in Section 4.1.4. The prototype realises

these rules by crossing out immutable features.

Section 6.2.4 of this case study demonstrates the use of the rights management by modelling the concrete extension with limited access rights. Only the necessary interface and interface specification is visible, as well as the affected cross-tree constraints. None of the rights management rules of Section 4.1.4 are violated by adding the concrete online radio station extension of this case study. Any other model modification (changing the interface specification, deleting or removing features) would violate the defined rules.

In consequence, the requirement to have a rights management implementation to support multiple contributors is fulfilled.

Compatibility with existing modelling strategies. The proposed POSF model is based on the concept of feature models. Removing the interface specifications leads to a model with a closed set of variability. Actually, this is a standard feature model. Even including concrete extensions can be considered as the addition of concrete features or closed variability. Hence, this modelling strategy is compatible with the existing feature modelling strategy. What is not supported is the open variability using interfaces and interface specifications. An extension set can be interpreted as a mandatory binary relation.

Low change impact. The change impact of the model in this case study can be analysed using the results of Chapter 5. The key intention of a low change impact level is to avoid modification interference between the different model contributors.

One suggestion of Section 5.2 is to avoid unnecessary cross-tree constraints between the main model and interface specifications. According to the online radio stations, this suggestion is obeyed. To use online radio, an internet connection is required. Instead of setting an implication from each radio station (which is an extension based on the station interface specification), the parent *Online Radio* feature implies the selection of the *Internet* feature. Hence, this constraint is limited to the main model.

One further advantage of the rights management is the opportunity to identify affected users for each potential model edit. The *Station Distributor* user is only able to create new radio station extensions without any cross-tree constraints. Thus, changes made by this user should have no effect on any other model contributor. On the other hand, every direct change according to the *Station* interface, its specification, the *Online Radio* feature, *Radio*, *Entertainment* and the root feature including all relations between the mentioned model elements would affect the *Station Distributor*.

Finally, the modelled case study offers a relatively low change impact by obeying the suggestions of Chapter 5. Even more importantly, the affected users can be identified as the potential impact of a change can be traced. Hence, the requirement for a low change impact can be considered as satisfied.

Human readable and computable. The human readability is given by the graphical notation of Section 6.1 and the visual appearance of the model using the prototype.

The computability is given by the implemented data structure as given in Section 4.2.2 (cf. Figure 4.9).

After performing this case study, it can be stated that all defined requirements are satisfied. This leads to the answer to the research question of this thesis. The question asks for a feasible variability modelling method for POSFs. As this case study shows, the proposed modification to feature models describes such a feasible variability modelling method.

6.4 Summary

The Chapters 4 and 5 proposed a modelling concept to support modelling Partly Open Software Families. To discuss the feasibility of this new concept, a case study has been used to check which of the former stated POSF modelling requirements are fulfilled.

To perform this evaluation, a concrete case study has been constructed in Section 6.1. The foundation of this case study was a car infotainment system based on the 2015 BMW 7 series. The modelling process has been done using the former developed prototypical modelling tool.

This case study's performance (cf. Section 6.2) realised the modelling of the feature structure, the introduction of interfaces including their specifications and the addition of cross-tree constraints. Multiple users organised in user groups were added, too, including the desired feature assignment to permit certain modification rights. Finally, a concrete extension based on one of the given interface specifications has been modelled. All modelling steps were based on the use cases the prototype was implemented for.

The evaluation in Section 6.3 checked the case study performance against the POSF requirements to show the actual feasibility of the proposed modelling concept.

The following last chapter will summarise this thesis and evaluate the presented work and its results based on the thesis' goals stated at the beginning. Furthermore, an outlook for further research work will be given to wrap up this thesis.

7 Conclusion and Outlook

The former chapters contained the main part of this thesis. The first part introduced the idea of Partly Open Software Families (POSFs) as a software family concept between Software Product Lines and Software Ecosystems (Chapters 1 and 2). As a modelling concept shall be developed, a detailed analysis was done afterwards that derived specific modelling requirements for those POSFs and analysed existing modelling strategies regarding these requirements to determine a feasible foundation to develop an appropriate modelling concept (Section 2.2.1 and Chapter 3). The modelling concept has been presented afterwards including the development of a prototypical modelling tooling and a change impact analysis (Chapters 4 and 5). The contribution ends with a feasibility check by means of a case study and its discussion according to the prior stated modelling requirements (Chapter 6).

This last chapter summarises the work and derives conclusions out of it based on the goals of this thesis (cf. Section 1.2). This conclusion is followed by an outlook with possible tasks for further research that has to be done to integrate the presented variability *modelling* concept into a viable variability *management* concept for POSFs.

7.1 Conclusion

Section 1.2 at the beginning of this thesis formulated a red thread for this thesis by giving a concrete research question and deriving goals from this question. Thus, the conclusion of this thesis evaluates the work done according to the accomplishment of the stated goals.

The research question is the following:

> *What variability modelling concept can be used to allow variability management of Partly Open Software Families as a concept placed between Software Product Lines and Software Ecosystems?*

This thesis offered a modified feature model as a new POSF modelling concept to answer this question. To get to this answer, the following major tasks were accomplished.

7.1.1 Comparison of SPLs and SECOs

The comparison of the two existing software family concepts has been done within the first part of Chapter 2. The concepts of Software Product Lines (SPLs) and Software Ecosystems (SECOs) were presented within Section 2.1.2 and 2.1.3.

An important aspect of these concepts is the supplier network with the contributing parties in the context of the customers and their concrete products. It is conspicuous that – in both cases – the resulting product is customised, but the customisation process is done at different times. In case of SPLs, the customisation is done on the vendor's side based on a closed set of variability. In case of SECOs, the customisation is done on the customer's side after getting the software platform from a platform vendor.

The actual comparison of both concepts, SPL and SECO, has been performed in Section 2.1.4.

7.1.2 Definition of the Concept between SPL and SECO

Based on the commonalities and differences of SPLs and SECOs a third type of software family was introduced in the second part of Chapter 2. A clear definition is proposed, which has been derived from the the results of the former SPL and SECO comparison.

The new concept was illustrated as a merged supplier network combining the SPL approach of a customised product from a software vendor and the SECO approach that allows further customisation on the customer's side based on an open set of variability.

Additionally, concrete modelling challenges were identified and, thus, POSF modelling requirements were fixed for the further work. Section 2.2.2 argued, why such a POSF concept offers chances and opportunities for customers and developers on both sides.

7.1.3 Analysis of Existing Variability Modelling Methodologies

Chapter 3 accomplished the third goal of an analysis of existing modelling strategies regarding the feasibility to serve as a foundation for a new POSF modelling concept.

Therefore,

- Feature models,
- Decision models,
- Orthogonal Variability Models (OVMs) and the
- Common Variability Language (CVL)

were presented, characterised, compared and evaluated according to the former stated POSF modelling requirements. The result of this chapter and this goal was a design suggestion to chose feature modelling to serve as a foundation to build an appropriate POSF modelling strategy.

7.1.4 Creation of a Feasible Variability Modelling Concept

A major goal of the thesis is the proposal of a new modelling strategy for POSFs. This goal has been accomplished within Chapter 4. Based on the requirements stated in Section 2.2.1 and the analysis result of Chapter 3, the challenges were derived that have to be tackled to get to an appropriate modelling strategy based on feature models (cf. Section 4.1). The following steps were accomplished afterwards:

- An interface notation element has been introduced to feature models to allow the support for open variability. Alongside with this notation element, the structural interface specification has been introduced, too.
- To support a graphical representation of interfaces with their specification and a distinguishable notation of concrete extensions that obey the regarding specification, graphical notation elements were introduced.
- Next to structural interface specifications, semantic information needs to be modelled as well. To support this, multiple levels of specification detail were discussed. Suggestions were given describing when to decide for which semantic representation and how to check whether an extension respects that specification or not.

- To allow multiple contributors to work with one POSF model, a rights management has been proposed with access rules for feature visibility and modification rights. The rights management is implemented by assigning users or groups of users to features. The model structure derives the visibility and modification rules based on stated rules for that.

To proof the feasibility of the proposed modelling strategy, a prototypical implementation of a modelling tool was given in Section 4.2 as well as a case study with a detailed discussion in accordance to the modelling requirements in Chapter 6.

The tool has been developed using EMF and the EMF Client Platform (ECP) based on an Ecore model that comprised the model elements as proposed in Section 4.1. The case study is based on multiple sources but uses a 2015 BMW 7 series as a foundation, because of its broad variety of different infotainment features and opportunities to integrate open variability.

7.1.5 Change Impact Analysis

One of the POSF requirements is a low change impact. This can be reduced to the question of what can be done to minimise the change impact. The intention is to avoid or at least reduce the modelling interference between multiple contributors. Furthermore, no external contributor should be able to invalidate a whole POSF model by developing an extension.

Therefore, Chapter 5 defined a procedure to systematically analyse the change impact. In a first step (cf. Section 5.1.1), all possible changes were identified and an analysis structure was derived from that. In a second step (cf. Section 5.1.2), the potential impact of each of the identified changes has been analysed. The third and last step of the analysis filtered the potential contributor interference (cf. Section 5.1.3).

Based on this analysis, the conclusion is that nearly every model modification can potentially lead to interference with other contributors. Thus, regulations are needed to reduce this effect. One major advantage of using a modelling strategy that supports a rights management and implements a user and user group management is the possibility to identify the users that are affected by a change. Hence, the potential impact of changes can be traced to seek for features with assigned users to identify a set of users that may be affected by a model change. Additionally, cross-tree constraints should be avoided between the main feature model and concrete extensions, because they contain the risk to invalidate an entire model as a worst case scenario.

7.2 Outlook

This thesis introduced a new concept of software families as a promising approach to create software that is highly customisable on a vendor's side (as known from SPLs) but afterwards on the customer's side as well (as known from SECOs). The concept combines the advantages of closed and open variability modelling by offering a tradeoff between community-driven openness and variability control by the main developing party. Ideally, the result is a highly customisable product that fits the customer's needs as good as possible while increasing the ROI for the software vendor as a result of a high customer satisfaction ratio.

Based on this software family concept, this thesis proposed a way to model software variability by multiple contributors while supporting closed and open variability at the same time and inside of one single model.

Variability modelling is part of the variability management that has to be done alongside the development lifecycle. Further work has to be done here to elaborate a full variability

management concept for POSFs. Especially the variability realisation aspect has to be discussed here. The proposed model already introduced the conceptual basis for that.

During the work for this thesis, the concept of MSPLs seems to be related to the POSF concept. This relation should be discussed in further research as part of the mentioned full variability management strategy. The basic intentions of both concepts are different: MSPLs introduce a compositional approach for product lines, whereas POSFs introduce open variability modelling. However, both concepts overlap on a conceptual level. Perhaps both concepts can be merged in a way that is advantageous for both sides. Further scientific discussion should clarify this.

When working on a feasible variability management strategy, discussing software evolution processes is necessary, too. The proposed modelling concept concentrates on variability in space (structural aspects). Discussing software evolution requires to cope with variability in time.

As soon as a variability management concept is ready for an actual test run in a real world project, a real world feasibility evaluation should be discussed in a scientific research context to evolve and improve the concept, optimise the modelling process and build a developer-friendly modelling toolset. The aim would be the development of a modelling and variability management strategy that may be considered by a broad range of software engineers to increase the potential customer satisfaction and the vendor's ROI.

A Task Description

This appendix includes the whole official task description that has been worked out in advance of this thesis. The task description comprises a thesis title proposal, the context and motivation of the work, the goals to achieve when elaborating the thesis and some planning attempts, namely a working outline and a schedule.

A.1 Title

The title of the intended work is proposed to be

Conceptual Variability Management in Software Families with Multiple Contributors

A.2 Context and Motivation

The world of software families is basically divided into Software Product Lines (SPLs) and Software Ecosystems (SECOs) which both follow different ideologies. Whereas in SPLs the variability space of derivable products is closed and mostly contributed by one party, variability in SECOs is mainly driven by a (public) community and – in consequence – follows an open approach.

While the transformation of software between both worlds is a frequently discussed topic (especially the question of how to evolve a software family from an SPL to a SECO), the aim of managing software families between both concepts mentioned seems to be out of scope of most discussions yet. Nevertheless, the idea of such software families makes a promising impression, as it would allow a very flexible, collaborative engineering and development of software families. A software family that is partly an SPL and a SECO finally is partly closed and open at the same time regarding the offered variability space.

One field in which such a concept seems to be promising is the area of in-car software that allows users to install apps to customise the user experience and to cope with differently timed product life cycles. Actual demand seems to exist as well, as in 2013 Schultis et al. (from Siemens) described the problem of handling industrial software products that have been designed as typical product lines, but with external influences.

Finally, a major problem in the described field is an appropriate and feasible variability modelling as part of the variability management respecting the existence of multiple contributors, as both concepts, SPLs and SECOs, seem to follow opposite approaches of centralised versus decentralised variability management. Finding a solution here is an essential part to realise a software family that combines the concepts of SPLs and SECOs. It is necessary to specify which part of the offered variability is closed and which part follows an open approach. It is required to clarify the meaning of *closed* and *open* in this case of a software family. Generally, it has to be possible to specify, which contributor is allowed to perform which modification to the variability model. The prevailing concepts do not respect this aspect so far.

A.3 Goals

The main task of the proposed thesis is to analyse possible concepts to realise variability modelling that allows developing software families that combine both closed and open variability approaches as known from SPLs and SECOs. To fulfil this task the thesis should accomplish the sub-goals described in the following.

A.3.1 Comparison of SPLs and SECOs

Current situation. Identify conceptual similarities of SPLs and SECOs regarding variability modelling with special focus on the contributing parties, their interrelation and coordination approaches. Where are the substantial differences to cope with when finding a common model?

To accomplish this task, a concrete distinctive definition of both SPLs and SECOs is needed, respecting the different ways of handling collaboration that exist in each of both software family concepts. Furthermore, the comparison should highlight characteristics of SPLs and SECOs that are desirable for the intended combined software family concept. Unwanted or even contradictory characteristics has to be identified here as well.

A.3.2 Definition of the Concept Between SPL and SECO

Scope setting. Compile a concrete definition of a partly open software family concept as a combination of SPLs and SECOs. This definition is missing yet but necessary to build up a common context to talk about.

To accomplish this task, an idea should be drafted that combines the desired characteristics of both SPLs and SECOs. Undesired characteristics has to be avoided and to ensure feasibility, contradictory characteristics must not be included at all. There has to be a clear distinction between the new concept and SPLs as well as SECOs. The need for that new concept should be justified exemplarily to clarify when the new concept would be appropriate where an SPL or a SECO would not.

A.3.3 Analyse Existing Variability Modelling Methodologies

Identify the concrete problem. Evaluate existing variability modelling methodologies regarding suitability and feasibility for partly open software families as previously defined.

Accomplishing this task requires a beginning inventory of existing variability modelling concepts available for SPLs and SECOs. To what extent are these methodologies already feasible regarding a partly open software family? What changes would be necessary to get to a feasibly variability modelling method?

A.3.4 Creation of a Feasible Concept

Solution proposal. This is the major goal of the thesis. Create a feasible draft for a variability modelling concept for partly open software families. This concept comprises a proof in the form of a prototypical implementation and a case study to demonstrate the feasibility.

Ideally, the task can be accomplished by combining promising and non-contradictory variability modelling concepts discussed in the previous goal and adding the necessary modifications to make it feasible. To proof the feasibility, a case study has to be created as an exemplary partly open software family with multiple contributors. The proof itself should be performed using a prototypically implemented tool that realises the developed variability modelling method.

A.3.5 Change Impact Analysis

How to continue. Finding answers to these questions: In how far are other contributing parties affected by changes that one party makes on the variability model? (Especially: What changes can be performed? What is the influence scope of these changes? Identify potential overlap with the action scope of other contributors.) Can this impact be avoided or at least reduced – how? Furthermore: Identify steps to take to get to a feasible concrete variability modelling methodology for partly open software families. What compromises have to be made or (at least) taken into account?

A.4 Working Outline

As the goals described in the previous section basically describe a sequential path for developing a variability modelling technique for partly open software families, the working outline is structured equally sequential. The working process is proposed to be structured as follows.

1. **Comparison of SPLs and SECOs** **(6 %)**
    a) Create common definitions for SPL and SECO
    b) Identify desired characteristics of SPLs and SECOs
    c) Identify undesired/contradictory characteristics of SPLs and SECOs
    d) *Writing (Comparison)*
2. **Defining the concept between SPL and SECO** **(6 %)**
    a) Generate idea concept
        i. Combine desired SPL/SECO characteristics
        ii. Feasibility proof (avoid contradictory characteristics)
    b) Clarify distinction to SPL and SECO
    c) Justify exemplarily the need for that concept
    d) *Writing (Definition)*
3. **Analyse existing variability modelling methodologies** **(10 %)**
    a) Research existing variability modelling methods
    b) Analyse feasibility of these techniques for new concept
    c) Identify necessary modifications of these techniques
    d) *Writing (Analyse existing VM)*
4. **Creation of a feasible concept** **(48 %)**
    a) Generate feasible idea
        i. Combine existing variability modelling concepts
        ii. Extend concept by required modifications
        iii. *Writing (Idea)*
    b) Conception of prototypical tool

c) Create exemplary case study
d) Writing (Concept and Case Study)
e) Create a prototypical tool
f) Run case study using tool
g) *Writing (Implementation and Run)*

5. **Change Impact Analysis** **(16 %)**

a) Analyse change impact
   i. What changes can be made?
   ii. Identify influence scope
   iii. Identify potential overlap with other contributors
b) Find ideas to reduce/avoid change-impact on other contributors
c) Identify steps to get to feasible concrete variability modelling methodology
d) *Writing (Change Impact Analysis)*

The major working packages are strictly based on the defined goals. The emphasised writing has to be done in parallel to the other tasks in the same group. Each task has been scheduled as shown in the Gantt charts of last section of this task description.

The project duration is fixed to 23 weeks, respectively 161 days. The percentage next to each working package group mentioned above shows the relative duration based on these 23 weeks. In total, 86 % of the project duration has been planned. The remaining 14 % of the available time will be used as a buffer and for thesis finalisation steps.

A.5 Schedule (Gantt-Chart)

The shown Gantt charts in Figures A.1, A.2 and A.3 visualise the project structure and schedule. The left part of Figure A.1 shows a zoomed-out overview whereas all other charts are detailed viewports of the whole chart representing one major work package each.

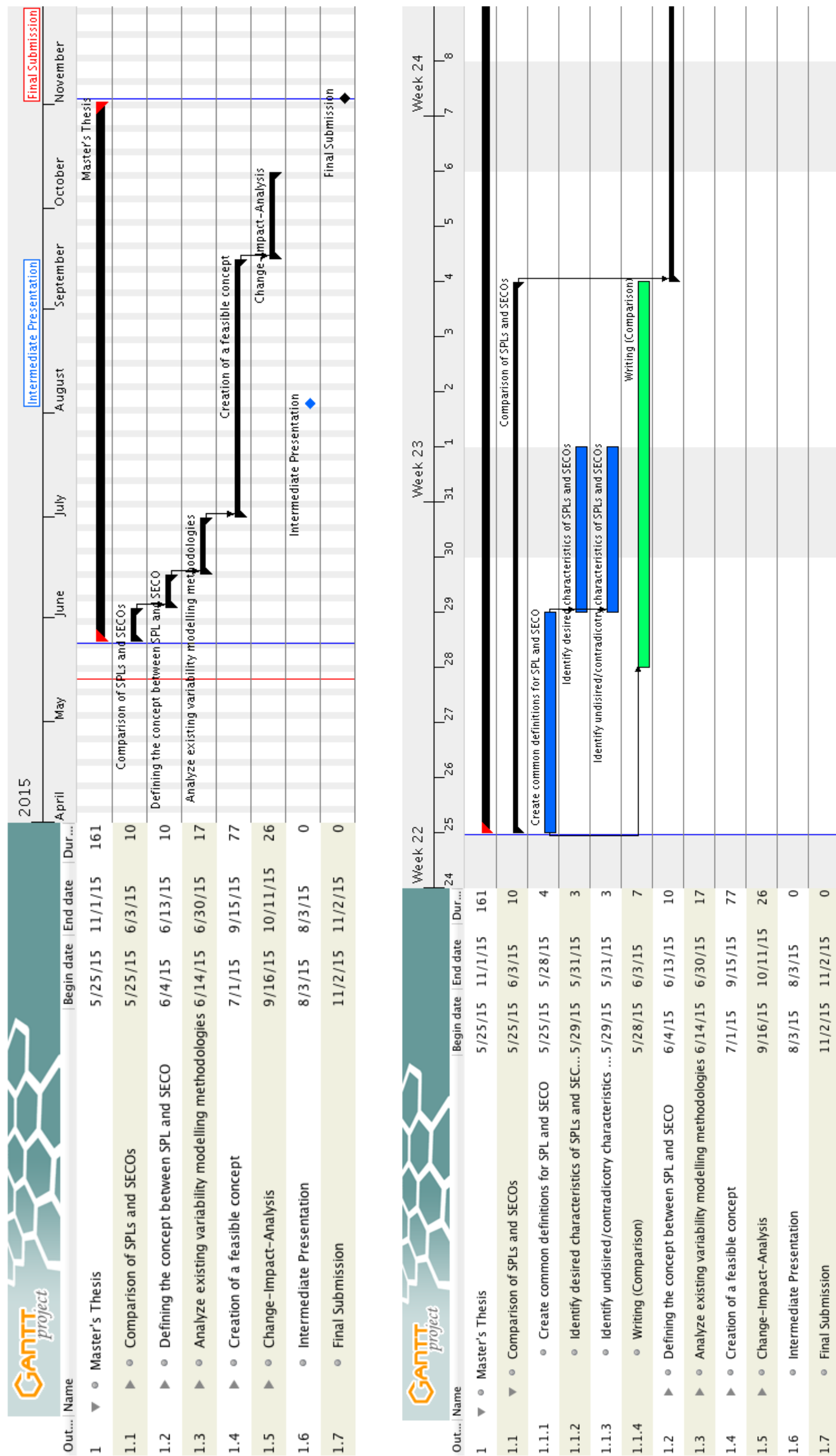


Figure A.1 – Gantt charts 1 and 2 out of 6

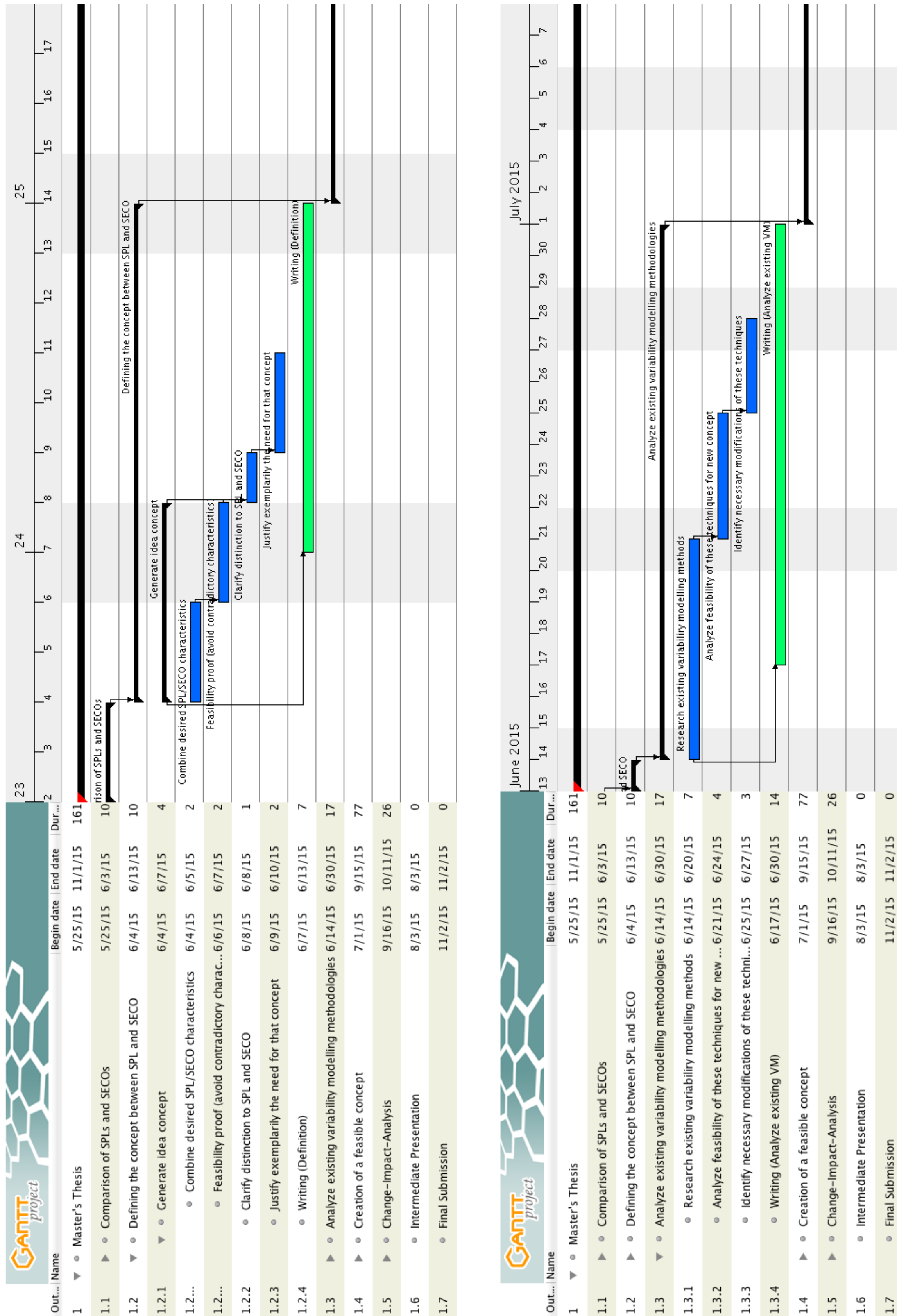


Figure A.2 – Gantt charts 3 and 4 out of 6

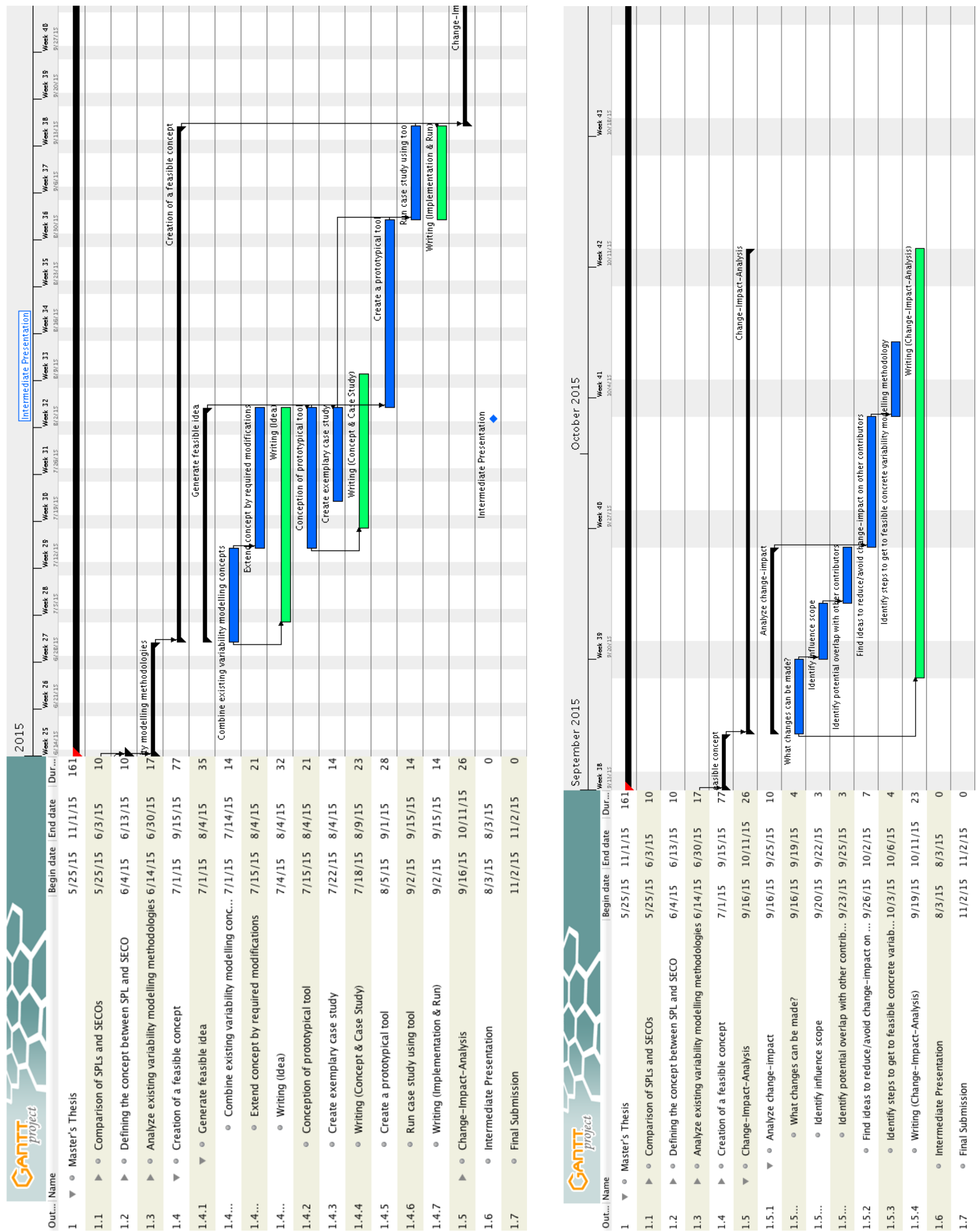


Figure A.3 – Gantt charts 5 and 6 out of 6

B Prototype Manual

This appendix provides a short user guide regarding the developed modelling tool of the second part of Chapter 4.

B.1 How to Install and Run the Tool

To execute the prototypical tooling, there are a few steps that have to be performed in advance to set up the software properly using the Eclipse Modeling Framework (EMF) and EMF Client Platform (ECP). In this guide, the following prerequisites are assumed:

1. A current Java Runtime Environment (JRE) and Java Development Kit (JDK) distribution is installed.[1]
2. The Eclipse IDE is installed, containing the current version of EMF and ECP. Ideally, the already fully-equipped Eclipse distribution *Eclipse Modeling Tools*[2] should be used.
3. Both necessary project folders

   `de.davidgollasch.posf.model` and `de.davidgollasch.posf.model.edit`

   are available to import them in the next step.

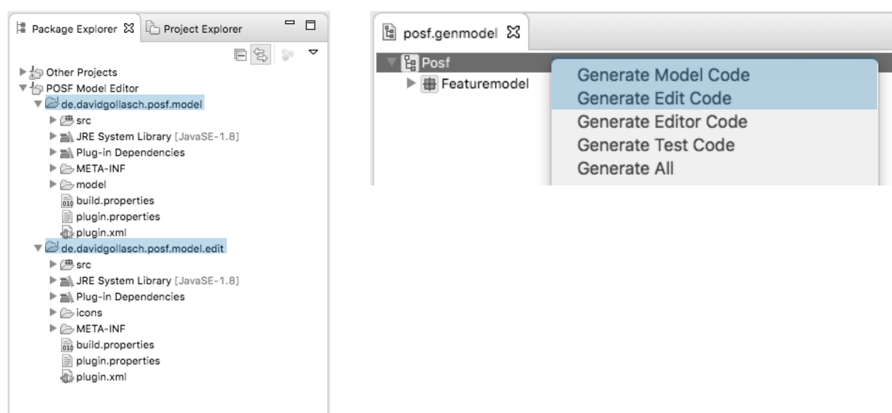


Figure B.1 – Setting up the prototype project using the Eclipse IDE

As a first step, the given project folders – as named above – need to be imported as projects into the Eclipse workspace environment. As soon as this is done, the Project Explorer shall show both projects similar to that given in the left part of Figure B.1.

To run the prototype, the source code needs to be generated using the underlying Ecore model. This is the consequence of using EMF as a foundation to create a modelling data structure.

[1]Download Java from `https://www.java.com/de/download/`

[2]Download EMT from `http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/mars1`

Initially, by importing the given project folders, the required source code is already given in a fully-generated form. However, in case any problem occurs during execution or changes to the model are performed, the model needs to be regenerated. This can be done by opening up the file

```
de.davidgollasch.posf.model/model/posf.genmodel
```

and right-clicking the shown root node `Posf`. The context menu shown in the right part of Figure B.1 appears. To generate the required code, both options *Generate Model Code* and *Generate Edit Code* have to be triggered.

Caution! Never delete the existing code files (*.java). As these files contain flagged non-generated code, deleting the files would remove the modifications, which are required for the tooling to run properly.

Finally, to run the tooling, an appropriate *Run Configuration* has to be set up. To do so, in the *Run Configurations* dialogue a new configuration needs to be created inside the *Eclipse Application* node on the left side.

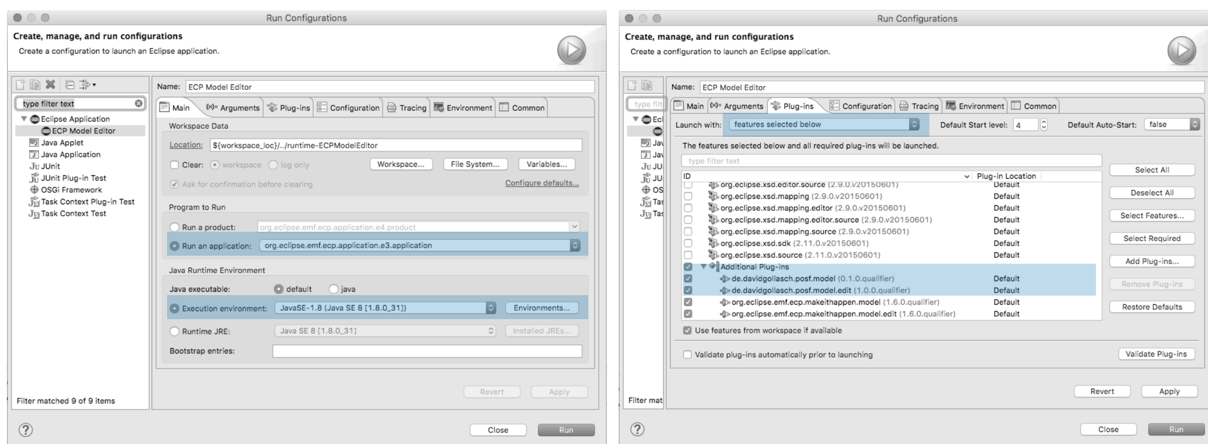


Figure B.2 – Creating a run configuration to execute the tooling

To create an appropriate configuration, some changes need to be made in the *Main* tab and the *Plug-ins* tab.[3]

- In the *Main* tab (cf. left dialogue of Figure B.2), the *Program to Run* has to be set to an application of the type *org.eclipse.emf.ecp.application.e3.application*. Secondly, ensure that the *Java Runtime Environment* is set to the current execution environment.

- In the *Plug-ins* tab (cf. right dialogue of Figure B.2), the correct selection is crucial for a proper run. To properly select the right bundles, *Launch with* has to be set to *features selected below* and all bundles should be initially deselected. Afterwards, the bundle *org.eclipse.emf.ecp.demo.e3.feature* has to be selected and an additional click on *Select Required* finalises the selection of required plug-ins. Additionally, the two former imported projects *...posf.model* and *...posf.model.edit* need to be added via clicking on *Add Plug-ins*.

Triggering the created run configuration starts the tooling as intended.

[3] Additional information regarding the ECP configuration can be found at `http://eclipsesource.com/blogs/tutorials/getting-started-with-the-emf-client-platform/`

B.2 How to Create a New Model Project

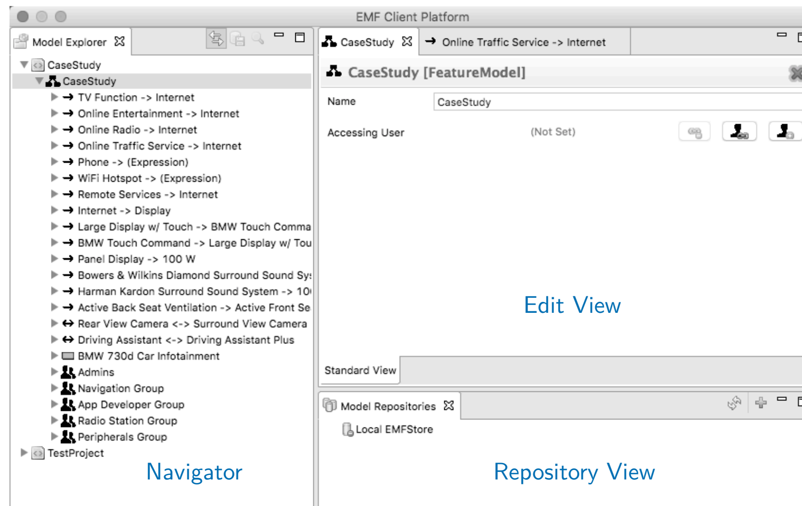


Figure B.3 – Prototype: GUI structure (navigator, edit view and repository view)

Starting the tooling leads to a GUI similar to Figure B.3. The interface provides three basic parts:

Navigator The navigator comprises all models and their containment structures, including features, different types of relations, constraints and user groups with users inside. The modelling is basically performed using this part of the window.

Edit View The upper-right part of the window contains a tabbed environment to show an editor for each object that is shown in the navigator. Each editor is context sensitive, so that it shows the elements of each object that can be modified only.

Repository View The repository view allows accessing various persistence layers, depending on which layers are implemented. For the purpose of this prototype, this part of the screen is not relevant.

To create a new model, the first and second step shown in Figure B.4 have to be executed. Thus, creating a new model requires the creation of a new project, which is done by choosing *Create new project* out of the context menu of the navigation view. The *Create Project* dialogue will open immediately. This dialogue shall be filled obeying the following steps:

- The *Provider* shall be an *Eclipse Workspace*.
- The assigned project name will be the name of the project node as shown in the navigator.
- To create a new project, *Create empty project* has to be selected.
- It is mandatory to select a *Filename*, which is the path to an **.xmi* file which will contain the model.
- Every model (or project) needs exactly one root node. To create a POSF model, the *Root Class* to choose is *http://de/davidgollasch/posf/FeatureModel*.

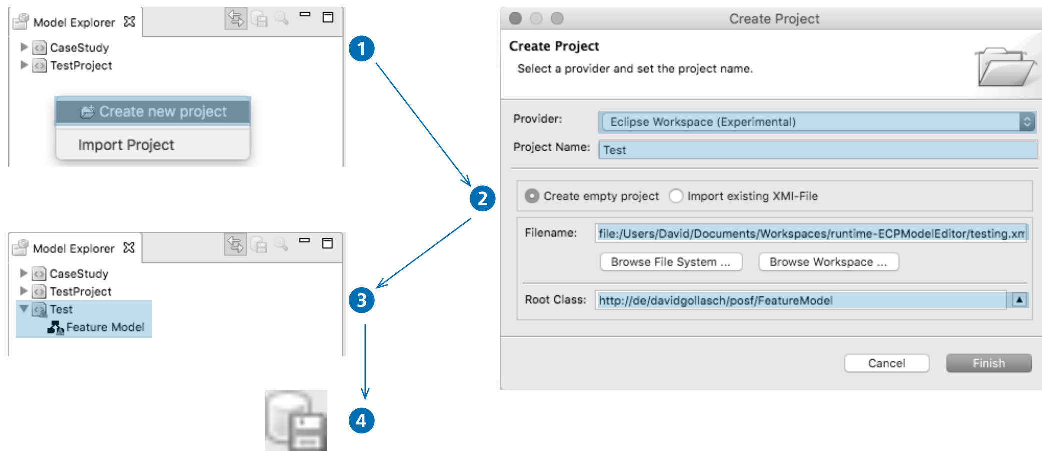


Figure B.4 – Creating a new model

If these steps are performed correctly, the navigator will show the just created project including the chosen root element of the model (cf. step 3 of Figure B.4). Double-clicking onto the *Feature Model* node opens the edit view, which allows assigning a name to the model.

Remark! When making changes to the model, saving it is required. This can be triggered using the save icon on top the the *Model Exporer* as shown in step 4 of Figure B.4.

B.3 How to Model

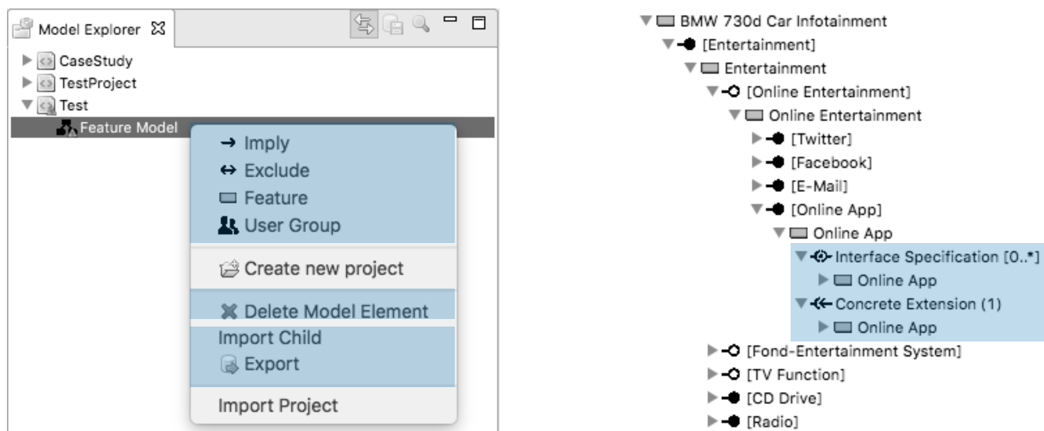


Figure B.5 – Modelling via the navigator's context menu; Exporting and importing models and model elements; Interface notation of the tooling

The whole modelling process in terms of creating model objects is done using the navigation view and context-sensitive right-click menu as shown in the left part of Figure B.5.

The modelling process starts with the root element. Each right-click on any of model elements shows the currently addable model elements according to the underlying model as described in Chapter 4. Basically, the following rules apply:

Feature Model The feature model itself allows adding one model root feature, several constraints (as implications or exclusions) and user groups.

Features Each feature may contain different relations, such as a mandatory, optional or set relation. Each of these relations comprises one or many further features. Mandatory and optional relations are binary relations, set relations represent groups. Furthermore, features can serve as an interface, which requires adding an interface specification into a feature. If a feature serves as an interface, a set of concrete extensions may be added as well.

Interfaces Interfaces are represented by means of an interface specification and – optionally – a set of concrete extensions as child elements of a feature. Interface specifications may contain a feature that serves as a root for that specification. This defines the root element for later to be developed extensions. Additionally, an interface specification is allowed to comprise constraints. The set of concrete extensions is allowed to comprise features representing a concrete extension each. Interfaces are represented as exemplarily shown in the right part of Figure B.5.

Constraints Constraints can be defined as child-nodes of the model's root element or inside an interface specification. Each constraint may contain a left and a right expression. Each expression can be a feature reference or a logical operator (*and*, *or*, *not*) that may contain further expressions until the leaf elements of a constraint are feature references only.

Users Users are organised in groups that can be defined as child-elements of the model's root element.

Next to adding new elements, the navigator's context menu offers the possibility to delete nodes of the model as well.

Remark! Remember to save the current state using the save button on top of the navigation view whenever changes to a model are performed.

B.4 How to Store/Export a Model

Storing a model can be performed using the *Export* option offered by the navigator's context menu, cf. the left part of Figure B.5.

Store a model To store or export a whole model, the root feature model element has to be selected before performing the *Export* option of the context menu.

Export an extension To export an extension, the context menu's *Export* option has to be performed onto a set of concrete extensions.

Remark! The prototype does not prevent users from exporting arbitrary parts of the model. Nevertheless, only the two actions above are explicitly intended to show, how working with POSF models using an appropriate tooling should look like.

B.5 How to Open a Model

There are two ways of opening a model. Both ways work rather equivalently.

Option 1 The model may be imported during the project creation process. This process is similar to the one of the new model creation, except that in the dialogue of Figure B.4 the option *Import existing XMI-File* needs to be selected. Hence, a formerly exported model may be imported inside of a new project.

Option 2 The alternative option is to use the *Import Project* option of the navigator's context menu. This leads to a dialogue which allows choosing an **.xmi* file that contains a modelling project.

B.6 How to Import a Partial Model

The process of importing partial models is required when desiring to import an externally developed (and formerly exported) concrete extension. Hence, this regards to the import of a partial model into a proper extension set element of the model.

The importing itself is done similar to opening a model using the secondly described way of using the navigator's context menu. Importing a partial model runs analogously but using the *Import Child* option of a concrete extension set's context menu.

Remark! Actually, the prototype does not prevent the user form importing any **.xmi* file at any location inside the model, unless the **.xmi* is not importable as it violates the data structure. Furthermore, there is no model validation available to check, whether the imported model obeys all modelling rules or not. In consequence, the user has to take care of this.

B.7 How to Apply User Perspectives

The prototype implements a dummy implementation of a rights management. This means, no user is actually prevented from performing modelling actions he would normally not be permitted to. The rights management is implemented in a way that crosses out all model elements where a user would be denied to get access to according to the presented modelling concept.

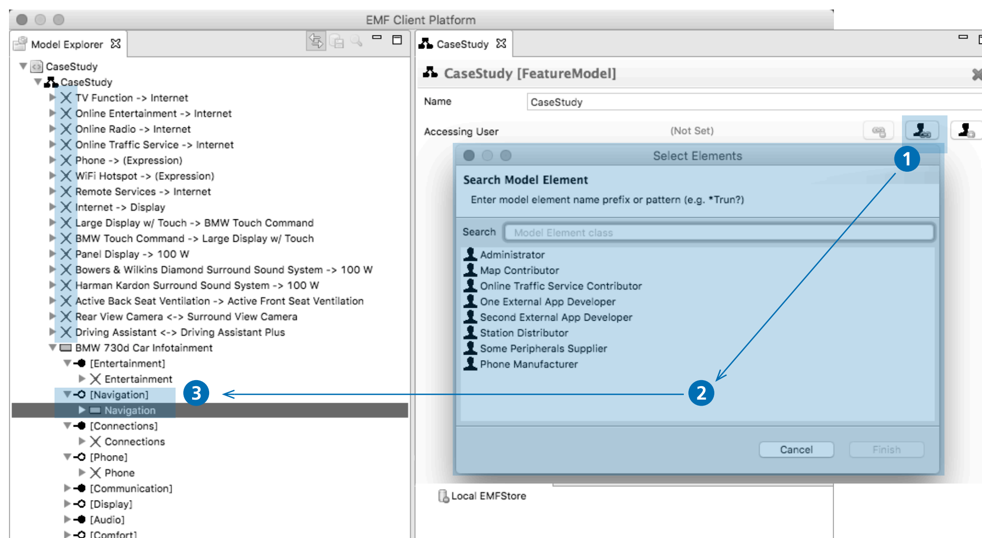


Figure B.6 – Setting user specific model perspectives

Firstly, introducing rights management requires the setup of user groups containing users and assigning users or user groups to the features, the access should be granted. Ideally, there is a user group for model administrators, which are permitted to get access to the whole model. Therefore, the root feature needs a user group assignment for that group of administrators. Usually, interfaces should be made visible for a selection of specific users or user groups. Therefore, the parent-features of the intended interface specification should be used to assign the desired user or group of users.

To invoke the rights management, a user has to be set as the currently accessing user. This can be done inside the edit view of the root feature model element as shown in Figure B.6. To update the view, the project needs to be closed and re-opened afterwards. This can be done as follows:

1. Choose *Close* from the project's context menu inside of the navigation view.
2. Double-click on the closed project.
3. Open the child-nodes of the feature model to review the result.

Navigating through the tree shows that only the path to the features is not crossed-out for which the access rule is given, including the whole model containment of these features and the constraints that affect any of the visible features. Users and user groups are always visible. Everything else is crossed-out.

C Case Study Construction

This appendix presents the compositional construction of the case study of a car infotainment system handled in this thesis. To generate a case study that is as realistic as possible, the car infotainment system of a 2015 BMW 7 series [BMW15a; BMW15b] shall be modelled – including the BMW Connected Drive system [BMW14]. Furthermore, to extend the model by supplemental infotainment related elements that are invisible to a customer's configuration process, additional sources (namely [Whi+10] and [HT08]) are used.

The presented tree models are not designed as a feature model yet but comprise all features that should be modelled in a structured way. The colour coding is done using the following semantics:

black The black feature is the model's root feature.

orange The orange features indicate the top-level structure, meaning the first level of categorisation of the given functionalities. They are chosen with separation of concerns in mind.

blue The blue features are based on the online car configurator from BMW for their 2015 BMW 7 series. [BMW15b; BMW15a; BMW14]

green The green-coloured features are added to implement further structuring and adding "hidden" features based on [Whi+10] and [HT08].

purple Purple-coloured features describe the future interfaces to open up the model in the sense of a Partly Open Software Family (POSF).

Figure C.1 gives a brief overview of the whole model that is presented below. Because it is too large for printing it entirely on one page in a readable way, the single parts of the model are presented in the following. This overview shows, how these model parts are composed.

Figure C.2 shows the top-level structure of the model, comprising the root feature *Car Infotainment System* and the sub-ordered categories. The categories either present hardware components of the system (e. g. *Connections*, *Dash Power Supply* and *Controls*) or offered functionality (e. g. *Communication*, *Entertainment* and *Navigation*).

The Figures C.3 to C.14 present the sub-trees of the top-level structure of Figure C.2. The interfaces *Online App*, *Station*, *Bluetooth Phone*, *USB Phone* and *Browser* include sub-trees with their interface structure that shall be modelled as an interface specification during performing the case study.

BMW 730d Car Infotainment
Entertainment
Online Entertainment
Fond–Entertainment System
TV Function
CD Drive
Online App
Twitter
Display Support
Connectivity
Online Mode
Offline Mode
Head–Up Display
Rear
Instruments
Panel
Facebook
E–Mail
Radio
Online Radio
DAB Tuner
FM Tuner
Station
Low Speed
High Speed
Navigation
Navigation System Professional
Real–Time Traffic Information
Online Traffic Service
Maps
North America
Europe
South America
Africa
Additional Map
Asia
Australia
Connections
USB
Diagnostics
Data Transfer
Bluetooth
Peripherals Protocol
Phone Protocol
Audio Protocol
Card Slot
RFID
WiFi
Phone
Phone Support w/ Wireless Charging
Specific Phone Support
Bluetooth Phone
Contacts
Calling
USB Phone
Contacts
Calling
Default Bluetooth Phone Support
Fond Wireless Phone Support
Communication
Internet
Browser
Panel Display Support
Rear Display Support
Default Browser
ConnectedDrive Services
Intelligent Emergency Call
WiFi Hotspot
TeleServices
Concierge Services
Remote Services
Display
Multi–Function Instruments Display
BMW Head–Up Display
Panel Display
Large Display w/o Touch
Large Display w/ Touch
Audio
Bowers & Wilkins Diamond Surround Sound System
Harman Kardon Surround Sound System
HiFi Speakers
Comfort
Active Front Seat Ventilation
Front Seat Heater
Active Back Seat Ventilation
Pre–Heating
Ambient Air Package
Parking
Parking Assistant
Rear View Camera
Rear Sensors
Front Sensors
Surround View
Controls
BMW Gesture Control
Panel Controls
Gestures
BMW Touch Command
Armrest Controls
Driving
BMW Night Vision w/ Pedestrian Recognition
Tyre Pressure Display
Driving Assistant Plus
Driving Assistant
Dash Power Supply
100 W
50 W

Figure C.1 – Case study structure – overview

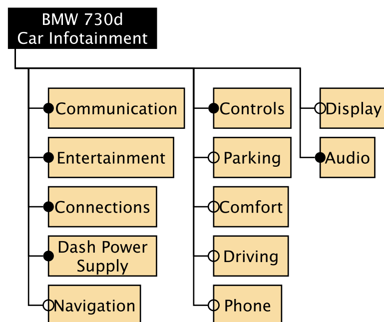


Figure C.2 – Case study structure – top-level structure

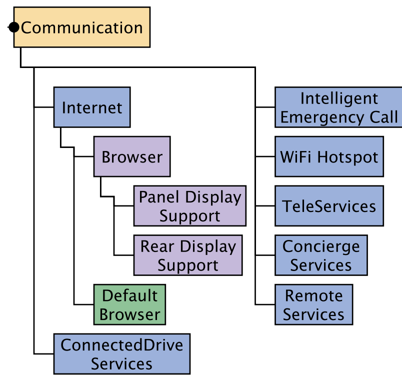


Figure C.3 – Case study structure: Communication

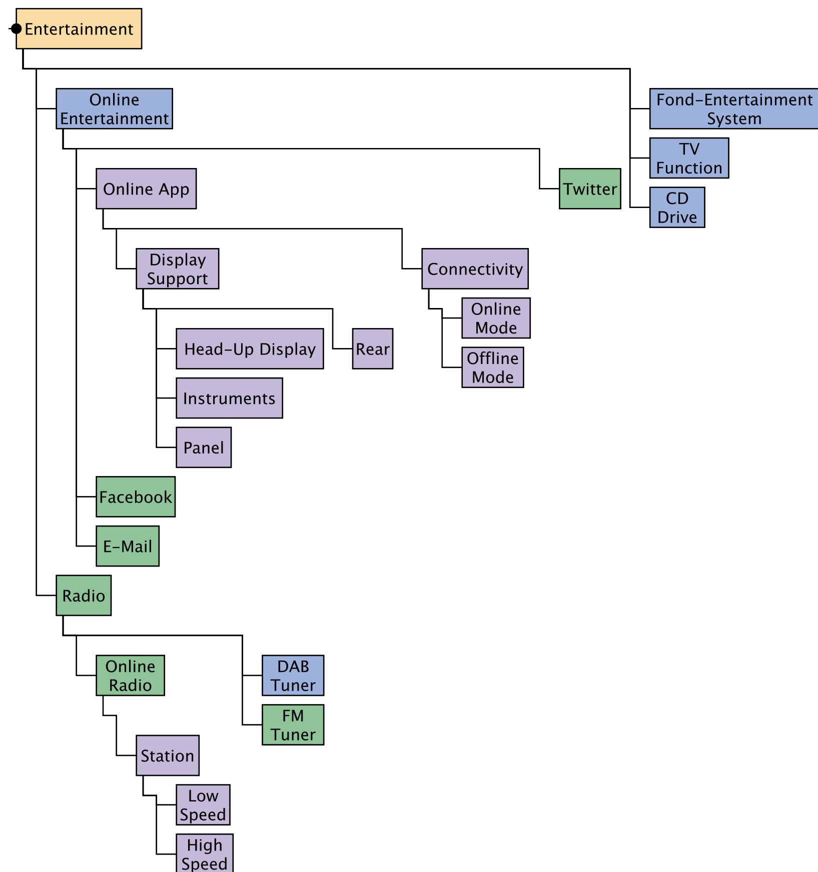


Figure C.4 – Case study structure: Entertainment

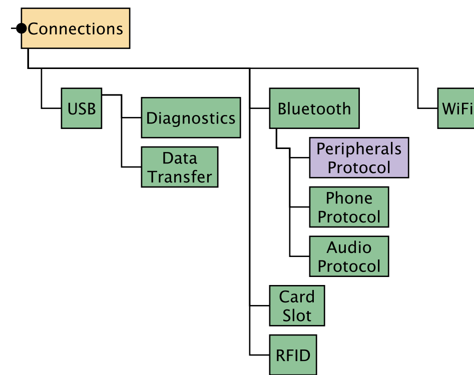


Figure C.5 – Case study structure: Connections

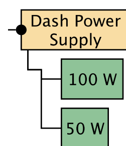


Figure C.6 – Case study structure: Dash Power Supply

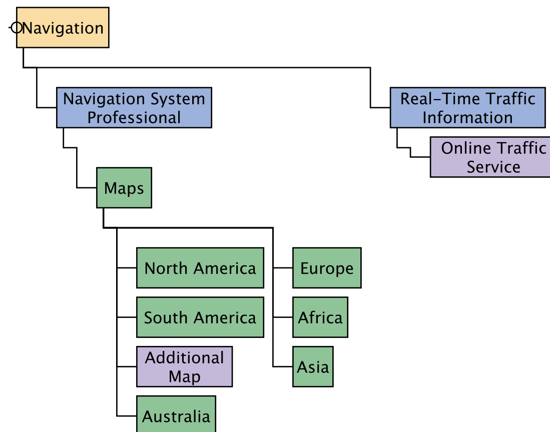


Figure C.7 – Case study structure: Navigation

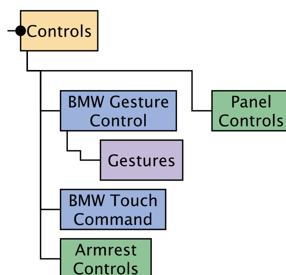


Figure C.8 – Case study structure: Controls

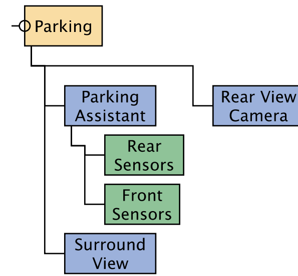


Figure C.9 – Case study structure: Parking

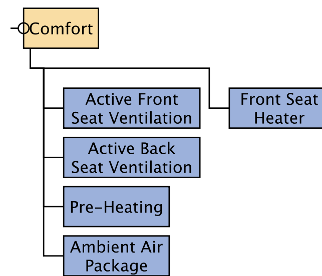


Figure C.10 – Case study structure: Comfort

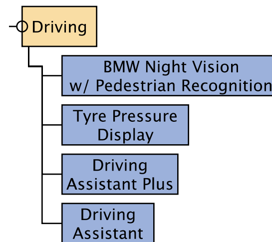


Figure C.11 – Case study structure: Driving

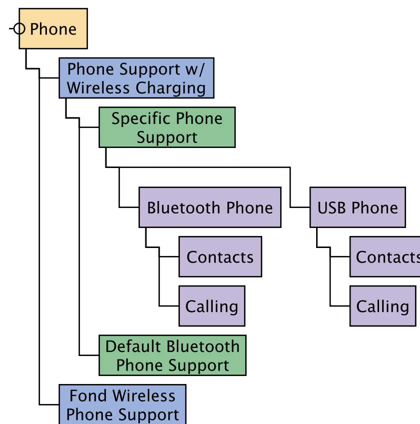


Figure C.12 – Case study structure: Phone

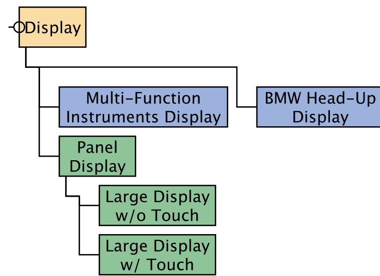


Figure C.13 – Case study structure: Display

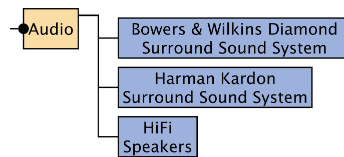


Figure C.14 – Case study structure: Audio

List of Figures

1.1 Venn diagram visualisation of SPLs, SECOs and POSFs 2

2.1 SAP Business ByDesign configurator screenshot 9
2.2 Two-lifecycle process of SPLE 9
2.3 SPL supplier network 10
2.4 SECO supplier network 11
2.5 Pie chart visualisation of SPLs, SECOs and POSFs 15
2.6 POSF supplier network 16
2.7 Requirements for a modelling strategy for POSFs 17

3.1 Model presentation aspects 22
3.2 Exemplary feature model for a car infotainment system; Kang et al. notation elements for feature models 22
3.3 Feature modelling meta model 24
3.4 Exemplary cardinality-based feature model 24
3.5 Exemplary attributed feature model fraction 25
3.6 Exemplary hyper feature model fraction 25
3.7 Exemplary DMN model showing the basic notation elements and the relation to BPMN 26
3.8 OVM notation elements 28
3.9 Exemplary OVM model representing a car infotainment system 29
3.10 CVL notation elements 30
3.11 CVL example showing a car infotainment example 31
3.12 Workflow for the models' feasibility analysis 33

4.1 Proposed POSF notation elements 44
4.2 Notation of interface specifications and concrete extensions 46
4.3 Levels of different semantics notation approaches 46
4.4 Visualisation of the rights management rules 50
4.5 Exemplary overview: POSF modelling 52
4.6 Use cases for the prototype (UML use case diagram) 54
4.7 Data structure abstraction for POSF feature models (UML class diagram) 55
4.8 GUI mockup for the prototype 56
4.9 Graphical representation of the underlying Ecore model 58
4.10 Final GUI of the prototype 59

5.1 Process visualisation of the change impact analysis 62
5.2 Graph of potential model changes 63

6.1 Case study structure – top-level structure 72
6.2 Case study – derived feature model part: Communication 75
6.3 Case study – derived feature model part: Entertainment 75

6.4 Case study – derived feature model part: Connections . . . 76
6.5 Case study – derived feature model part: Dash Power Supply . . . 76
6.6 Case study – derived feature model part: Navigation . . . 76
6.7 Case study – derived feature model part: Controls . . . 77
6.8 Case study – derived feature model part: Parking . . . 77
6.9 Case study – derived feature model part: Comfort . . . 77
6.10 Case study – derived feature model part: Driving . . . 77
6.11 Case study – derived feature model part: Phone . . . 78
6.12 Case study – derived feature model part: Display . . . 78
6.13 Case study – derived feature model part: Audio . . . 78
6.14 Case study – concrete extension to the Station interface . . . 81
6.15 Modelling a concrete online radio station extension using the prototype . . . 82

A.1 Gantt charts 1 and 2 out of 6 . . . v
A.2 Gantt charts 3 and 4 out of 6 . . . vi
A.3 Gantt charts 5 and 6 out of 6 . . . vii

B.1 Setting up the prototype project using the Eclipse IDE . . . ix
B.2 Creating a run configuration to execute the tooling . . . x
B.3 Prototype: GUI structure (navigator, edit view and repository view) . . . xi
B.4 Creating a new model . . . xii
B.5 Modelling using the prototype . . . xii
B.6 Setting user specific model perspectives . . . xiv

C.1 Case study structure – overview . . . xviii
C.2 Case study structure – top-level structure . . . xviii
C.3 Case study structure: Communication . . . xix
C.4 Case study structure: Entertainment . . . xix
C.5 Case study structure: Connections . . . xx
C.6 Case study structure: Dash Power Supply . . . xx
C.7 Case study structure: Navigation . . . xx
C.8 Case study structure: Controls . . . xx
C.9 Case study structure: Parking . . . xxi
C.10 Case study structure: Comfort . . . xxi
C.11 Case study structure: Driving . . . xxi
C.12 Case study structure: Phone . . . xxi
C.13 Case study structure: Display . . . xxii
C.14 Case study structure: Audio . . . xxii

List of Tables

2.1 Comparison of SPLs and SECOs 12
2.2 Characteristics of POSFs 16

3.1 Example of a decision model based on the car infotainment example 27
3.2 Brief summary of the presented modelling methodologies 32
3.3 Feasibility overview of feature models 35
3.4 Feasibility overview of decision models 36
3.5 Feasibility overview of orthogonal variability models 38
3.6 Feasibility overview of the Common Variability Language 39
3.7 Consolidated feasibility overview for all presented modelling techniques 40

4.1 Fulfillment of the POSF requirements – comparison of feature models and the new POSF-enabled feature models 51

6.1 Case study – cross-tree constraints 73
6.2 Case study – users and user groups 73

List of Abbreviations

BPMN	Business Process Modelling Notation	26
CBSE	Component-Based Software Engineering	48
COTS	Commercial Off-The-Shelf	13
CVL	Common Variability Language	30
DAB	Digital Audio Broadcasting	23
DMN	Decision Model and Notation	26
ECP	EMF Client Platform	56
EMF	Eclipse Modeling Framework	56
ERP	Enterprise Resource Planning	8
FM	Frequency Modulation	23
GUI	Graphical User Interface	55
HFM	Hyper Feature Model	25
HUD	Head-Up Display	23
IDE	Integrated Development Environment	10
IDL	Interface Description Language	48
IMAP	Internet Message Access Protocol	7
JDK	Java Development Kit	ix
JRE	Java Runtime Environment	ix
LCD	Liquid-Crystal Display	52
MIDL	Microsoft Interface Definition Language	49
MMS	Multimedia Messaging Service	28
MPL	Multi Product Line	33
MSPL	Multi Software Product Line	33
OCL	Object Constraint Language	48
OMG	Object Management Group Inc.	26
OOP	Object-Oriented Programming	48
OVM	Orthogonal Variability Model	28
PLE	Product Line Engineering	24
POP	Post Office Protocol	7
POSF	Partly Open Software Family	2
ROI	Return on Investment	1

SaaS Software as a Service 8
SAT Satisfiability 34
SECO Software Ecosystem 1
SDK Software Development Kit 18
SMART Specific, Measurable, Achievable, Relevant and Time-Bound 47
SMS Short Message Service 28
SPL Software Product Line 1
SPLE Software Product Line Engineering 7
SVN Apache Subversion 14
UML Unified Modelling Language 31
USB Universal Serial Bus 72
VSpec Variability Specification 30

Contents of Electronic Medium (DVD)

- Contents of Electronic Medium as HTML File .... `/Contents.html`
- List of Useful Web Links .... `/Links.html`
- Thesis Text Files .... `/Thesis`
- Source Code and Executables of Software Artefacts .... `/Software`
  - Models .... `/Software/Models`
  - Source Code and Executables .... `/Software/Source`
  - Case Study .... `/Software/CaseStudy`
- Used References in Electronic Form (e. g. Adobe PDF) .... `/References`
- Presentation Slides (Defence and Intermediate Presentation) .... `/Slides`

Bibliography

[App07] Apple Inc. *Apple Reinvents the Phone with iPhone*. Jan. 9, 2007. URL: https://www.apple.com/pr/library/2007/01/09Apple-Reinvents-the-Phone-with-iPhone.html (visited on 06/18/2015) (cit. on p. 1).

[App15] Apple Inc. *Absatz von Apple iPhones weltweit vom 3. Geschäftsquartal 2007 bis zum 3. Geschäftsquartal 2015 (in Millionen Stück)*. 2015. URL: http://de.statista.com/statistik/daten/studie/12743/umfrage/absatz-von-apple-iphones-seit-dem-jahr-2007-nach-quartalen/ (visited on 10/09/2015) (cit. on p. 1).

[BA96] Shawn A. Bohner and Robert S. Arnold. *Software change impact analysis*. Los Alamitos, Calif: IEEE Computer Society Press, 1996. 376 pp. (cit. on p. 61).

[Ber+14] Thorsten Berger, Rolf-Helge Pfeiffer, Reinhard Tartler, Steffen Dienst, Krzysztof Czarnecki, Andrzej Wasowski, and Steven She. "Variability Mechanisms in Software Ecosystems". In: *Information and Software Technology* (2014) (cit. on p. 1).

[BKP04] Günter Böckle, Peter Knauber, and Klaus Pohl. *Software-Produktlinien: Methoden, Einführung und Praxis*. Ed. by Klaus Schmid. Heidelberg: Dpunkt.verlag, 2004 (cit. on p. 1).

[BMW14] BMW AG. *BMW Connected Drive. (product information)*. 2014 (cit. on pp. 71, xvii).

[BMW15a] BMW AG. *7 (product information)*. 2015 (cit. on pp. 71, xvii).

[BMW15b] BMW AG. *BMW Konfigurieren (730d)*. 2015. URL: http://www.bmw.de/vc/ncc/xhtml/start/startEngineSelectionWithParams.faces?referer=topnav&productType=1&brand=BM&market=DE&country=DE&modelRangeId=G11 (visited on 09/29/2015) (cit. on pp. 71, xvii).

[Bog05] Robert Bogue. *Use S.M.A.R.T. goals to launch management by objectives plan - TechRepublic*. TechRepublic. Apr. 25, 2005. URL: http://www.techrepublic.com/article/use-smart-goals-to-launch-management-by-objectives-plan/ (visited on 10/12/2015) (cit. on p. 47).

[Bos09] Jan Bosch. "From Software Product Lines to Software Ecosystems". In: *Proceedings of the 13th International Software Product Line Conference*. Pittsburgh, PA, USA: Carnegie Mellon University, 2009, pp. 111–119 (cit. on p. 1).

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative programming: methods, tools, and applications*. Boston: Addison Wesley, 2000. 832 pp. (cit. on p. 22).

[Chu03] Steven Church. *Impact Analysis: An Essential Part of Software Configuration Management*. 2003. URL: http://www.stickyminds.com/sites/default/files/article/file/2013/XUS2650642file1_0.pdf (cit. on p. 61).

[DPA15] DPA. *Porsche schmeißt Android raus: Macht Google unsere Autos zu rollenden Datenkraken? - FOCUS Online*. FOCUS Online. Oct. 19, 2015. URL: http://www.focus.de/auto/news/porsche-schmeisst-android-raus-google-android-auto-schnueffelt-keine-fahrzeug-daten-aus_id_5022254.html (visited on 10/21/2015) (cit. on p. 20).

[Ent15] Entrepreneur Staff. *Return on Investment (ROI) - Small Business Encyclopedia*. Entrepreneur.com. 2015. URL: http://www.entrepreneur.com/encyclopedia/return-on-investment-roi (visited on 10/09/2015) (cit. on p. 13).

[Gol13] David Gollasch. "Qualitätssicherung mittels Feature-Modellen". Bachelor Thesis. Dresden: TU Dresden, 2013 (cit. on p. 34).

[Hau+08] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K. Olsen, and Andreas Svendsen. "Adding Standardized Variability to Domain Specific Languages". In: IEEE, Sept. 2008, pp. 139–148. DOI: 10.1109/SPLC.2008.25. (Visited on 08/13/2015) (cit. on p. 30).

[HT08] Herman Hartmann and Tim Trew. "Using Feature Diagrams with Context Variability to Model Multiple Product Lines for Software Supply Chains". In: IEEE, Sept. 2008, pp. 12–21. DOI: 10.1109/SPLC.2008.15. (Visited on 09/25/2015) (cit. on p. xvii).

[HW12] Øystein Haugen and Andrzej Wasowski. "CVL Mini-Tutorial". Common Variability Language Wiki. Oct. 24, 2012. URL: http://www.omgwiki.org/variability/doku.php (cit. on p. 30).

[Kan+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Carnegie-Mellon University Software Engineering Institute, Nov. 1990 (cit. on p. 22).

[Lan+08] Jan Christian Lang, Thomas Widjaja, Peter Buxmann, Wolfgang Domschke, and Thomas Hess. "Optimizing the supplier selection and service portfolio of a SOA service integrator". In: 41st Hawaii International Conference on System Sciences. Darmstadt: Inst. für Betriebswirtschaftslehre, 2008 (cit. on p. 10).

[Li+13] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. "A survey of code-based change impact analysis techniques: A SURVEY OF CODE-BASED CIA TECHNIQUES". In: *Software Testing, Verification and Reliability* 23.8 (Dec. 2013), pp. 613–646. DOI: 10.1002/stvr.1475. (Visited on 09/29/2015) (cit. on p. 62).

[Lin+04] Frank van der Linden, Jan Bosch, Erik Kamsties, Kari Känsälä, and Henk Obbink. "Software Product Family Evaluation". In: *Software Product Lines*. Ed. by Robert L. Nord. Red. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, and Gerhard Weikum. Vol. 3154. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 110–129. (Visited on 06/22/2015) (cit. on p. 5).

[Lin07] Frank van der Linden. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. In collab. with Klaus Schmid and Eelco Rommes. Berlin ; New York: Springer, 2007. 333 pp. (cit. on p. 9).

[MK14] Mohamed Mussa and Ferhat Khendek. "Acceptance Test Optimization". In: *System Analysis and Modeling: Models and Reusability*. Ed. by Daniel Amyot, Pau Fonseca i Casas, and Gunter Mussbacher. Vol. 8769. Cham: Springer International Publishing, 2014, pp. 158–173. (Visited on 10/12/2015) (cit. on p. 48).

[Obj14] Object Management Group Inc. *Decision Model and Notation - Version 1.0 FTF convenience document (clean)*. Nov. 2, 2014. URL: `http://www.omg.org/spec/DMN/1.0/Beta2/PDF` (cit. on pp. 26, 28).

[Par76] D.L. Parnas. "On the Design and Development of Program Families". In: *IEEE Transactions on Software Engineering* SE-2.1 (Mar. 1976), pp. 1–9. DOI: `10.1109/TSE.1976.233797`. (Visited on 06/24/2015) (cit. on p. 6).

[PBL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software product line engineering: foundations, principles, and techniques*. 1st ed. New York, NY: Springer, 2005. 467 pp. (cit. on pp. 1, 7 sq., 28).

[PM06] Klaus Pohl and Andreas Metzger. "Variability management in software product line engineering". In: ACM Press, 2006, p. 1049. DOI: `10.1145/1134285.1134499`. (Visited on 06/24/2015) (cit. on p. 28).

[RBR09] Fabricia Roos-Frantz, David Benavides, and Antonio Ruiz-Cortés. "Feature model to orthogonal variability model transformation towards interoperability between tools". In: *KISS@. ASE, New Zealand* (2009) (cit. on pp. 24, 28).

[RG99] Mark Richters and Martin Gogolla. "A Metamodel for OCL". In: *UML'99 — The Unified Modeling Language*. Ed. by Robert France and Bernhard Rumpe. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 1723. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 156–171. (Visited on 10/12/2015) (cit. on p. 48).

[RS10] Marko Rosenmüller and Norbert Siegmund. "Automating the Configuration of Multi Software Product Lines." In: *VaMoS* 10 (2010), pp. 123–130 (cit. on p. 33).

[Sch13] Klaus Schmid. "Variability Support for Variant-Rich Software Ecosystems". 4th International Workshop on Product Line Approaches in Software Engineering. San Francisco, California, USA, May 20, 2013 (cit. on p. 1).

[SS13] Klaus Schmid and Eduardo Santana de Almeida. "Product Line Engineering". In: *IEEE Software* 30.4 (July 2013), pp. 24–30. DOI: `10.1109/MS.2013.83`. (Visited on 07/01/2015) (cit. on p. 9).

[SSA14] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. "Capturing variability in space and time with hyper feature models". In: ACM Press, 2014, pp. 1–8. DOI: `10.1145/2556624.2556625`. (Visited on 05/30/2014) (cit. on p. 25).

[TBK09] Thomas Thüm, Don Batory, and Christian Kästner. "Reasoning about edits to feature models". In: IEEE, 2009, pp. 254–264. DOI: `10.1109/ICSE.2009.5070526`. (Visited on 06/25/2013) (cit. on p. 68).

[TJM96] Mitchell M. Tseng, Jianxin Jiao, and M. Eugene Merchant. "Design for Mass Customization". In: *CIRP Annals - Manufacturing Technology* 45.1 (Jan. 1996), pp. 153–156. DOI: `10.1016/S0007-8506(07)63036-4`. (Visited on 06/28/2015) (cit. on p. 8).

[Whi+10] J. White, D. Benavides, D. C. Schmidt, P. Trinidad, B. Dougherty, and A. Ruiz-Cortes. “Automated diagnosis of feature model configurations”. In: *Journal of Systems and Software* 83.7 (2010), pp. 1094–1107. DOI: 10.1016/j.jss.2010.02.017. (Visited on 09/25/2015) (cit. on p. xvii).

Confirmation

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, November 2, 2015