



Vrije Universiteit Brussel

Faculty of Science & Bio-Engineering Sciences
Department of Computer Science
Web & Information Systems Engineering Lab (WISE)

The Feature Assembly Approach for Modelling & Knowledge Management of Software Variability

Thesis submitted in fulfilment of the requirements for the award of the degree of Doctorate in Science

Lamia A. M. Abo Zaid

Academic year: 2012 - 2013

Promoter: Prof. Dr. Olga De Troyer



© 2013 Lamia A.M. Abo Zaid

All rights reserved. No parts of this dissertation may be reproduced or transmitted in any form, electronic, mechanical, photocopying, recording, or any other means without written permission from the author.

Acknowledgments

This research would not have been possible without the help of many individuals and organizations. Therefore, in these lines I would like to thank all the people who in many different ways have made this work possible and turned it into a wonderful experience.

I would like to express my deep and sincere gratitude to my promoter, Prof. Dr. Olga De Troyer, head of the Web & Information Systems Engineering (WISE) Laboratory of the Vrije Universiteit Brussel. Olga was always available for advise, guidance, encouragement, and fruitful stimulating discussions. I believe that our discussions have largely enhanced my scientific experience and allowed introducing a new feature modelling language and modelling technique presented in this thesis. I would also like to sincerely thank her for her thorough and careful review of this thesis. I would like to thank her for supporting, and encouraging me to participate in several conferences worldwide. I learned a lot from her about life, research, how to tackle new problems and how to develop techniques to solve them. Many thanks Olga.

Next, I would like to thank the members of my jury: Prof. Dr. Ir. Geert-Jan Houben, Prof. Dr. Frederik Gailly, Prof. Dr. Inge van de Weerd, Prof. Dr. Eddy Van Dijck, Prof. Dr. Beat Signer, and Prof. Dr. Viviane Jonckers, for being members of the jury evaluating this dissertation. Their comments have improved the text and are greatly acknowledged.

In addition, I have been very privileged to get to know and to collaborate with many other great people. Thanks go to my ex-colleague at WISE, Dr. Frederic Kleinermann. The many conversations I had with him on various topics gave me valuable insights and triggered new ideas. Thanks also go to all my colleagues and ex-colleagues at the WISE, namely Mohammed El Dammagh, William Van Woensel, and Sven Casteleyn for our interesting discussions about many different things.

I would also like to thank my master thesis student Tom Puttemans and my bachelor student Jasper Tack for their implementations which were a great help for validating my work. Thank you Tom and Jasper.

A special thanks goes to my master thesis promoter Prof. Geert-Jan Houben who gave me the opportunity to work in the VariBru research project while he was at the Vrije Universiteit Brussel before heading off to TU Delft, many thanks Geert-Jan.

I would like to thank Innoviris, the Brussels institute for research and innovation (www.innoviris.be) for funding the VariBru project (www.varibru.be) that provided the context for this PhD work.

I would also like to thank Sebastien Le Grand and Frederic Arijs from Antidot for their valuable comments on the Feature Assembly approach being a pilot user for the approach.

I would like to thank the VariBru team members for providing a nice and friendly atmosphere in which ideas were discussed. Particularly, I would like to thank Wim Codenie, Tom Tourwé, and Nicolás González-Deleito from Sirris for our discussions about variability problems.

My deep gratitude and sincere thanks goes to my father for his motivation and support throughout my studies, from kindergarten to university. Without his fine education and thorough support, I would probably not have taken up the eagerness or motivation to do this PhD. I would like to sincerely thank my mother for her support, encouragement and love, which gave me so much joy. I would also like to thank my sisters for their support and encouragement.

I would like to sincerely thank my husband for his love, support, encouragement, and patience during the PhD period. I thank him for being there for me in times when I needed him most which gave me the power to perform this research. I also would like to thank my two lovely daughters Rasha and Rawan; with their lovely smiles they do miracles.

Finally, I would like to thank all the people I forgot to mention here. Many people contributed in some way to this PhD and I thank all of them.

Abstract

Over the last decades software development has evolved into a complex task due to the large number of features available in software, the many feature interactions, the distributed nature of software, and the many stakeholders involved. At the same time, there is more demand to deliver software rapidly while maintaining customer intimacy. This situation has led to the emergence of so-called Software Product Lines (SPL) or more generally *variable software*. SPLs tend to *manufacture* the software development process. Instead of developing a single product, the fundamental base is to develop multiple closely related but different products. These different products share some common features but each individual product has a distinguishable set of features that gives each product a unique flavour. Unfortunately, variability comes at the price of increasing complexity. The key issue for success is to have a balance between the added flexibility the variability offers and the complexity the variability brings to the development cycle. Therefore, there is a need for efficiently modelling and managing the knowledge around software variability as early as domain analysis during the domain engineering of the software product line. During domain analysis, variability modelling techniques allow to explicitly represent the variability and commonality of features while clearly indicating their influence on the complexity. Feature-oriented modelling techniques have been commonly used to model the variability and commonality in software product lines. Variability information management refers to the continuous management of the knowledge represented by the variability models (often involving many stakeholders) making this knowledge explicit and readily available.

In this thesis, we propose the Feature Assembly approach, a novel approach for modelling and managing knowledge about software variability. Feature Assembly also introduces the principle of combining reuse and variability. First of all, the Feature Assembly approach could help companies better define their products by thinking in terms of “features”. Furthermore, it allows companies to gradually introduce variability in their products and make use of previous modelled features. The Feature Assembly approach is a combination of the *Feature Assembly Modelling technique*, the *Feature Assembly Reuse Framework*, and the *Feature Assembly Knowledge Representation Framework*.

The backbone of the Feature Assembly approach is the *Feature Assembly Modelling technique*, which allows defining modular “parts” (i.e. features) via separating concerns and clearly distinguishing features that represent variability (i.e. variation points) from those which don’t. The modelling technique aims for keeping the created models simple and understandable

and for improving the extensibility of the features and feature models. The Feature Assembly Modelling technique is based on providing a set of easy to use, and unambiguously defined modelling concepts. Furthermore, it aims at scaling down complexity by promoting modelling with separation of concerns, because trying to deal with all possible viewpoints at the same time is very difficult and will usually result in badly structured models. A more scalable approach is to model the required capabilities of the software with respect to one viewpoint at the time. Therefore, in the Feature Assembly Modelling technique we model software from different viewpoints, called *perspectives*. Perspectives provide an abstraction mechanism which allows focusing on features that are related to a certain viewpoint.

Additionally, the *Feature Assembly Reuse Framework* introduces the idea of composing software by assembling features. Furthermore, it promotes feature reusability by storing features in a so-called Feature Pool, which acts as a feature repository for a company. The main idea is that new feature models can be made by combining existing features (stored in the feature pool) with new features. Newly defined features are then added to the Feature Pool, leading it to continuously grow. Such an approach allows companies to consider reuse as early as the design phase, therefore efficiently making use of previous experiences. In addition, reuse at the domain analysis level could encourage reuse at the architecture design and development levels, increasing the overall productivity and reducing development cost.

The *Feature Assembly Knowledge Representation Framework* offers management of the information contained in the defined Feature Assembly models. Explicitly representing and storing this information unlocks knowledge that would otherwise be stored in documentation and in people's minds. For this purpose, the Feature Assembly Ontology is defined. It acts as a formal documentation repository in which the information is stored. Users can readily retrieve this information at any point in time. Furthermore, the framework provides detection for modelling errors and conflicts by providing a validation of the correctness of the models.

Declaration

Parts of this thesis have been published:

Chapter 6:

Abo Zaid, L., Kleinermann, F., De Troyer, O. (2010). Feature Assembly Modelling: A New Technique for Modelling Variable Software, 5th International Conference on Software and Data Technologies, Proceedings Vol: 1, pp: 29 - 35, Eds. José Cordeiro Maria Virvou Boris Shishkov, Publ. SciTePress, ISBN 978-989-8425-22-5, Athens, Greece

Abo Zaid, L., Kleinermann, F., De Troyer, O. (2010). Feature Assembly: A New Feature Modeling Technique. In : 29th International Conference on Conceptual Modeling, Lecture Notes in Computer Science, Vol. 6412/2010, pp. 233-246, Springer. DOI:10.1007/978-3-642-16373-9_17

Chapter 7:

Abo Zaid, L., and De Troyer, O. (2011). Towards Modeling Data Variability in Software Product Lines, T. Halpin et al. (Eds.): BPMDS 2011 and EMMSAD 2011, LNBIP Vol 81, pp. 453-467. Springer, Heidelberg (2011). DOI:10.1007/978-3-642-21759-3_33

Chapter 9:

Abo Zaid, L., Kleinermann, F., De Troyer, O. (2011). Feature Assembly Framework: towards scalable and reusable feature models. In Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems (VaMoS '11). pp.1-9. ACM, New York, NY, USA. DOI:10.1145/1944892.1944893

Chapter 11:

Abo Zaid, L., and De Troyer, O. (2012). Modelling and Managing Variability with Feature Assembly – An Experience Report, Chaudron et al. (Eds.): EESSMod 2012 EESSMOD'12, ACM, October, Innsbruck, Austria. DOI:10.1145/2424563.2424575

Table of Contents

Acknowledgments	I
Abstract.....	I
Table of Contents	IV
List of Figures.....	XI
List of Tables	XV
List of Listings.....	XVII
List of Abbreviations.....	XIX
Chapter 1: Introduction	1
1.1 Research Context	2
1.2 Research Scope	4
1.3 Problem Statement	5
1.4 Research Questions	7
1.5 Positioning of the Research	8
1.5.1 Feature Modelling Methods	8
1.5.2 Feature Modelling for Large and Complex Systems	8
1.5.3 Reuse and Feature Modelling.....	9
1.5.4 Knowledge Management and Software Models	9
1.6 Research Approach and Methodology.....	10
1.7 Research Contributions	13
1.8 Thesis Outline.....	14
Chapter 2: Variability Modelling Using Feature Models	17
2.1 Software Variability	17
2.2 Software Variability Modelling.....	20
2.3 Feature Models	21

2.3.1 Normalizing Feature Models.....	26
2.4 Mainstream Feature Modelling Techniques	27
2.4.1 Feature-Oriented Reuse Method (FORM)	27
2.4.2 FeatureRSEB.....	28
2.4.3 van Gorp et al. Feature Graph.....	29
2.4.4 Riebisch et al. Feature Models.....	30
2.4.5 PLUSS.....	31
2.4.6 Cardinality Based Feature Models.....	31
2.5 Feature Modelling Methods based on UML.....	32
2.5.1 Clauss UML Variability Stereotypes	32
2.5.2 Ziadi et al. UML Variability Profile	33
2.5.3 Gomaa Variability Metaclasses.....	33
2.5.4 Korherr and List UML Variability Profiles	34
2.6 Summary	34
Chapter 3: Related Work.....	35
3.1 Representing and Analysing Feature Models	35
3.2 Feature Models for Configuration	37
3.3 Modelling with Separation of Concerns.....	38
3.4 Model Integration and Consistency Checking.....	40
3.5 Multiple Product Lines	41
3.6 Variability Modelling and Databases.....	43
3.7 Feature Model Visualization.....	44
3.8 Summary	44
Chapter 4: Background.....	47
4.1 Conceptual Modelling	48
4.2 Knowledge Representation Techniques	50
4.2.1 Logic Based Knowledge Representation	51
4.2.2 Semantic Networks	52
4.2.3 Ontologies.....	53
4.2.4 Rule-Based Knowledge Representation.....	55
4.3 Semantic Web Knowledge Management Techniques	55

4.3.1 OWL	56
4.3.2 Querying RDFs/OWL Ontologies.....	57
4.3.3 Reasoning on RDFs/OWL Ontologies.....	58
4.4 Knowledge Management Applied to Software Variability.....	59
4.5 Summary	60
Chapter 5: Challenges for Software Variability Modelling.....	61
5.1 Limitations of Mainstream Feature Modelling Techniques	62
5.1.1 Difficulties in Using the Feature Modelling Technique in Practice.....	63
5.1.2 Ambiguity in Modelling Concepts.....	65
5.1.3 Limited Reuse Opportunities	66
5.1.4 Lack of Abstraction Mechanisms.....	67
5.2 Challenges in Managing the Information in Feature Models	67
5.2.1 Information Management to Support Feature Modelling	68
5.2.2 Information Management of Feature Models	69
5.3 Recommendations for Feature Assembly.....	70
5.4 Summary	71
Chapter 6: The Feature Assembly Modelling Technique	73
6.1 Feature Assembly Overview	73
6.2 Running Example – E-Shop Product Line	74
6.3 Variability Analysis	75
6.4 Multi-Perspective Approach	77
6.4.1 System Perspective	79
6.4.2 User Perspective.....	81
6.4.3 Functional Perspective	82
6.4.4 Graphical User Interface Perspective.....	84
6.4.5 Goal Perspective	86
6.4.6 Non-Functional Perspective	87
6.4.7 Discussion.....	88
6.5 Feature Assembly Modelling (FAM) Language	91
6.5.1 Features	92
6.5.2 Feature Relations.....	93

6.5.3 Feature Dependencies	95
6.5.3.1 Feature dependencies within the same perspective	95
6.5.3.2 Feature dependencies between different perspectives	96
6.5.4 FAM Formal Specification	97
6.5.4.1 FAM Syntax.....	97
6.5.4.2 FAM Formal Semantics.....	101
6.6 Discussion	103
6.7 Summary	107
Chapter 7: Feature Assembly Modelling For Data Intensive Applications	109
7.1 The Persistent Perspective	110
7.1.1 Defining the Persistent Perspective.....	111
7.1.2 Refine the Persistent Perspective	112
7.2 Linking Feature Assembly Models and Data Models	113
7.2.1 The Centralized Data Model Approach	114
7.2.2 The Decentralized Data Model Approach.....	116
7.3 Summary	117
Chapter 8: The Quiz Product Line Case	119
8.1 Problem Statement	119
8.2 Feature Assembly Models for the QPL.....	121
8.2.1 QPL System Perspective.....	122
8.2.2 QPL Users Perspective.....	124
8.2.3 QPL Functional Perspective.....	125
8.2.4 QPL Graphical User Interface Perspective	131
8.2.5 Completing the Model	134
8.2.6 QPL Persistent Perspective	136
8.3 QPL Variable Data Model	139
8.4 Extensibility of the Feature Assembly Modelling Technique – An example.....	140
8.5 Lessons Learned	142
8.6 Summary	143

Chapter 9: The Feature Assembly Reuse Framework	145
9.1 Why Feature Assembly?	145
9.2 Overview of the Feature Assembly Reuse Framework	148
9.3 The Feature Pool	149
9.3.1 Feature Pool Example	151
9.4 Assembling Features with Feature Assembly	153
9.4.1 Feature Assembly Example	155
9.5 Summary	157
Chapter 10: Feature Assembly Knowledge Management Framework	159
10.1 Overview	160
10.1.1 Why OWL?	161
10.2 The FAM Ontology	162
10.2.1 The FAM Ontology Vocabulary	164
10.2.2 FAM Error Detection via the FAM Ontology	170
10.2.2.1 FAM Ontology - Error Capturing Rules	172
10.2.2.2 FAM Ontology - Error Debugging	173
10.2.3 Populating the FAM Ontology with Individuals	176
10.3 FAM Knowledge Manipulation	179
10.3.1 FAM Ontology Browsing	179
10.3.2 FAM Ontology Querying	180
10.3.3 Dedicated Ontology Browsing and Querying	182
10.4 The Feature Pool Ontology Representation	186
10.5 Summary	188
Chapter 11: Feature Assembly in Practice	191
11.1 Pilot Survey	191
11.2 ANTIDOT Experience Report	192
11.2.1 Method Adopted	193
11.2.2 Feature Assembly Modelling Technique	193
11.2.3 Feature Assembly Knowledge Manipulation	197
11.2.4 The Feature Assembly Reuse Framework	199
11.2.5 Discussion	200

11.3 Threats to Validity.....	200
11.4 Summary	201
Chapter 12: Conclusions and Future Work.....	203
12.1 Summary	203
12.1.1 Steps in the Research and Artefacts developed:.....	203
12.2 Contributions and Achievements.....	207
12.3 Limitations.....	208
12.4 Future Work.....	209
List of References.....	213
Appendix A: A Conceptual Model of Feature Mainstream Models.....	227
Appendix B: FAM Ontology in OWL Functional Syntax.....	231
Appendix C: OWL DL Description Logic Representation.....	237
Appendix D: Feature Pool Ontology in OWL Functional Syntax.....	239

List of Figures

Figure 1.1: Domain and Application engineering.....	1
Figure 1.2: Research Areas related to the Feature Assembly Approach.....	10
Figure 2.1 Sample of Quiz Product Line possible products.....	19
Figure 2.2: Feature Model of <i>Car</i> Product Line	24
Figure 2.3: Feature Model of <i>Car</i> Product Line	25
Figure 2.4 Possible normalization for a)optional b)mandatory alternative features	26
Figure 2.5 Possible normalization for a)optional b)mandatory OR features	26
Figure 2. 6 A feature model in FORM for the Private Branch Exchange (PBX)	28
Figure 2.7 A feature model in FeatureRSEB	29
Figure 2.8 van Gorp et al. feature graph for a mail client product line.....	30
Figure 2.9 Riebisch et al. feature model for a library Product line.....	30
Figure 2.10 PLUSS feature model for a Motor Engine System	31
Figure 2.11 CBFM feature model for an E shop product line	32
Figure 4.1: Conceptual Modelling Process	49
Figure 4.2: Sample Semantic Network	53
Figure 5.1 Feature Model of GPL, ambiguity in Graph Type definition.....	65
Figure 5.2 Example showing ambiguity of feature models	66
Figure 5.3 Example showing the impact of change in Feature Models	67
Figure 5.4 Example showing possible use cases for different stakeholders	69
Figure 6.1: Overview of the Feature Assembly Modelling process.....	74
Figure 6.2: Overview of the different perspectives used to model the E-Shop	78
Figure 6.3: Feature Assembly Perspective Selection Process.....	90
Figure 6.4: FAM Meta-Model.....	91
Figure 6.5: FAM feature notations: (a) Concrete Feature (b) Abstract Feature.....	92
Figure 6.6: FAM feature notations.....	94
Figure 6.7: FAM representation of the E-Shop's Order Process feature	94
Figure 6.8: Semantic Definition of a Modelling Language	97
Figure 6.9 FODA model of PBX problem	104

Figure 6.10 FAM model of PBX problem	105
Figure 6.11 FODA Representation of the GPL & FAM Representation of GPL.....	106
Figure 6.12 Support for changes and feature reuse a comparison	107
Figure 8.1: QPL System Perspective	123
Figure 8.2: QPL User Perspective.....	125
Figure 8.3: An excerpt of the QPL Functional Perspective	127
Figure 8.4: Functional Perspective - the Quiz-Question Assignment Feature.....	128
Figure 8.5: Functional Perspective - the Operational Settings Feature.....	129
Figure 8.6: Functional Perspective - the Answer Validation Feature	129
Figure 8.7: Functional Perspective - the Quiz Layout Feature	130
Figure 8.8: Functional Perspective - the Quiz Reporting Feature.....	130
Figure 8.9: GUI Perspective.....	132
Figure 8.10: GUI Perspective.....	133
Figure 8.11: GUI Perspective- Quiz Layout Feature	134
Figure 8.12: Persistent Perspective –Persistent QPL Feature	136
Figure 8.13: Persistent Perspective – Question Persistent Feature	137
Figure 8.14: Persistent Perspective – User Persistent Feature	138
Figure 8.15: Persistent Perspective – User-Quiz Info Persistent Feature	138
Figure 8.16: QPL Variable Data Model (Represented with EER).....	140
Figure 8.17: Cultural Perspective.....	141
Figure 9.1: Overview of the Feature Assembly Process	147
Figure 9.2: Feature Assembly Reuse Framework Overview	149
Figure 9.3: Feature Pool meta-model.....	150
Figure 9.4: Excerpt of the System perspective for the QPL	151
Figure 9.5: The Feature Pool Features Extracted from the QPL FAM.....	152
Figure 9.6: Feature Assembly Process	154
Figure 9.7: Feature Assembly Process	156
Figure 10.1: Overview of the Feature Assembly Knowledge Representation Framework	160
Figure 10.2: FAM Meta-Model.....	163
Figure 10.3: Corresponding FAM Ontology Meta-Model visualized by OntoGraf	164
Figure 10.4: FAM Ontology Class Hierarchy Shown in Protégé	169

Figure 10.5: Explanation For an Inconsistency Detected by The Reasoner	175
Figure 10.6: Explanation for a Cardinality Error- Using Protégé.....	175
Figure 10.7: An Excerpt of the QPL System Perspective.....	176
Figure 10.8: Example using of <i>Dependency_Reason</i> and the <i>Dependency_Owner</i> annotations in Protégé.....	179
Figure 10.9: Browsing the FAM Ontology for QPL.....	180
Figure 10.10: Querying for Features Using the OWL2Query Plugin in Protégé.....	181
Figure 10.11: The FAM Ontology browser visualizing the QPL	183
Figure 10.12: The <i>Perspectives</i> Tab.....	184
Figure 10.13: The <i>Options</i> Tab	184
Figure 10.14: The FAM Ontology browser’s Search facility.	185
Figure 10.15: The FAM Ontology browser’s Display facility.....	186
Figure 10.16: Feature Pool meta-model.....	187
Figure 10.17: FP Ontology in Protégé, Showing Usage of the Product Line Class. ...	188
Figure 11.1: Excerpt of the CMS specifications	194
Figure 11.2: Excerpt of Comments on CMS’s First Models	195
Figure 11.3: Screenshot showing how Information can be found in Feature Assembly Models - Applied to the models of Antidot.	198
Figure 11.4: Screenshot showing how Feature Assembly Models are visualized allowing users to interact with the information contained in the models - Applied to the models of Antidot.....	198
Figure 12.1: Overview of the work presented in this thesis in relation to our research questions.....	204
Figure A. 2 Conceptual model of feature models	227
Figure. A.3. Feature model showing shipping cost example	228
Figure. B.1. OWL DL Axioms and Facts	237
Figure. B.2 OWL DL	238

List of Tables

Table 2.1. Graphical Notation of Feature Types and Their Relations 23

Table 7.1: Relation between feature assembly model concepts and data modelling concepts..... 114

Table 7.2: Annotations denoting relations between features and database concepts... 114

List of Listings

Listing 2.1: Possible configurations of Car product line shown in figure 2.2	25
Listing 2.2: Possible configurations of Car product line shown in figure 2	25
Listing 6.1: FAM Feature Dependencies, notations and semantics.....	96
Listing 8.1: Feature Assembly-to-data model mappings	139
Listing 10.1: DL Axioms Representing the Feature Class.....	165
Listing 10.2: DL Axioms Representing the Feature Dependencies	166
Listing 10.3: DL Axioms Representing the Abstract Feature Class	166
Listing 10.4: DL Axioms Representing the Concrete Feature Class	167
Listing 10.5: DL Axioms Representing the Option Feature Class	167
Listing 10.6: DL Axioms Representing the Rules that derive the Variation Points and Variants	169
Listing 10.7: An Example Showing Possible Modelling Errors	170
Listing 10.8: An Example Showing a Cyclic Error	171
Listing 10.9: An Example Showing Redundant dependency.....	171
Listing 10.10: Rule to capture Inconsistency Error due to cyclic <i>uses</i> dependency	172
Listing 10.11: Rules to capture Inconsistency Errors due to conflicting dependencies	173
Listing 10.12: Rules to Capture Redundancy Errors Due to Redundant Dependencies	173
Listing 10.13: Rules to Capture Cardinality Errors	173

List of Abbreviations

SPL	Software Product Line
FAM	Feature Assembly Modelling
FD	Feature Diagram
FM	Feature Model
FODA	Feature Oriented Domain Analysis
FORM	Feature Oriented Reuse Method
GPL	Graph Product Line (a feature modelling benchmarking problem)
UML	Unified Modelling Language
OCL	Object Constraint Language
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
OWL	Web Ontology Language
SWRL	Semantic Web Rule Language
SPARQL	SPARQL Protocol and RDF Query Language (an RDF query language)
QPL	Quiz Product Line
SoC	Separation of Concerns

Chapter 1

Introduction

Increasing productivity, reaching more customers, and reducing costs are key aspects for the success of today's software development business. Furthermore, devices are increasingly becoming more software demanding, increasing the pressure on companies to deliver "quality" software at affordable prices and in a short time. One way to increase productivity and reduce the time/cost is by adopting "mass customization" and "mass development". This can be achieved via incorporating *variability*¹ early in the software development process, thus such software is known as *Variable Software* [Bosch, 2000].

Variable software has the ability to leverage the development of software from a single product approach to a product line approach. In a product line approach, the base is put for developing a set of *variable* assets which can be combined differently to make distinct products instead of just one product [Bosch, 2000] [Asikainen, 2004]. Often such variable software is known under names such as *software product family* or *software product line* [Bosch, 2000]. A software product line is commonly defined to consist of a common architecture, and a set of reusable assets. Together they are used systematically in producing individual products [Bosch, 2000]. The goal is to plan for the development of a set of closely related software products rather than for a single product. This enables an efficient reuse of assets during the development cycle, which is the main benefit of applying the product line technique. The product line is then *configured* to produce different products. Each product derivable from the product line encloses a different set of assets which makes it distinct. The process of producing different products from the product line is referred to as the *configuration* process. The engineering of a software product line is usually done in two phases: *domain engineering* and *application engineering* [Pohl et al., 2005] [Anquetil et al., 2010]. The purpose of domain engineering is to model the commonality and variability between the members of a software product line. Reusable assets are produced by domain engineering and then specialised during application engineering to derive the final products as shown in figure 1.1.

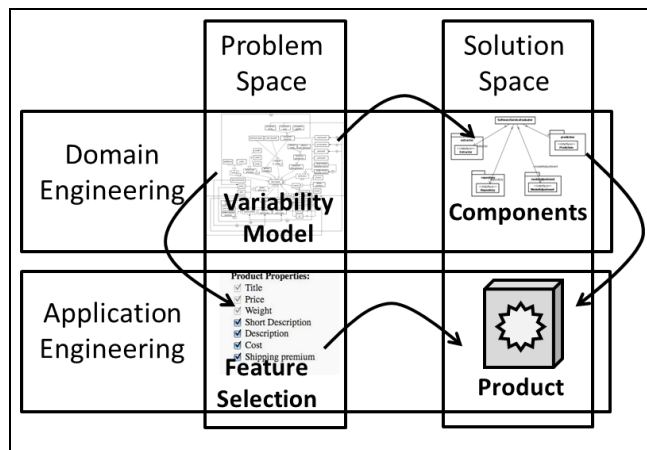


Figure 1.1: Domain and Application engineering, modified after [Pohl et al., 2005]

¹ *Variability* is the ability of a system to be efficiently extended, changed, customised or configured for use in a particular context [Asikainen et al., 2007].

The research on software product lines is driven by the increase of software demand, and accompanied with the large similarity in the software delivered to different customers and/or for different platforms. On-going research in the field of software variability includes topics that range from specifying and modelling software variability to actually implementing and configuring this variability.

This thesis is situated in the domain of variable software. However, this is a broad domain, therefore in the next section, section 1.1, the research context is described into more details and section 1.2 provides the actual scope of the thesis. Section 1.3 deals with the problem statement and section 1.4 provides the research questions. In section 1.5 we describe the position of our research with regard to other scientific work in the context of the research. The next section, section 1.6 elaborates on the research methodology used. Section 1.7 gives a summary of the research contributions and finally section 1.8 provides an outline of the thesis.

1.1 Research Context

An essential step to realize variability is the adequate planning of variability. Planning variability means carefully inspecting the domain of interest for information that allows identifying the commonality and variability between members of the product line. This implies properly understanding the domain needs and using the proper methods to represent this knowledge, and being able to communicate this knowledge to the different people involved at different moments in time. Such knowledge is not always straightforward or directly available; the modelling practice should help express this knowledge. It is not always easily expressible as the process involves many stakeholders with different concerns and speaking different languages (e.g., end users, customers, marketing specialists, domain engineers, research and innovation specialists, architecture engineers, project managers, etc.). Additionally, the modelling technique and the models themselves should provide support for practitioners to question their ideas and understanding of the system (or rather the system's domain) under consideration. Furthermore, the models should provide a medium for the different stakeholders to communicate their understanding of the different aspects of the represented information.

In order to characterize the variability and commonality for a certain system, first there is a need to explicitly identify the different characteristics of that system. In software product line engineering, the term “feature” is used to refer to a prominent characteristic or capability of a software system. Once these *features* are identified, it becomes important to distinguish which of these features are variable and which are not. *Variable features* are those that are optional to have in a product, i.e. they may exist in one or more products of the product line but not in all. Features that exist in all the products of the product line are referred to as *common features*. Variable features are associated with restrictions that govern their existence (or absence) in a certain product; this information is vital for producing feasible product configurations. Furthermore, there may be dependencies between features. Some features may be conflicting while others will work together to achieve the goals of the software. Therefore, it is also important to reveal the information on these feature interactions to allow safe configurations of products. Failing to do so will result in misconfigured products which are erroneous, inconsistent, and vulnerable.

Therefore, for the success of product line development, it is important to explicitly model the above-mentioned information, i.e. it is important to represent which features are variable and which features are common, the restrictions on feature selection, and the feature

interactions. The process of representing variability is referred to as *variability modelling*². Variability modelling techniques are usually based on *feature modelling* or *decision modelling* [Czarnecki et al., 2012]. Both approaches have a slightly different emphasis. Feature modelling approaches focus on commonality and variability modelling. Therefore features are first class citizens in the feature modelling techniques, and result in a *feature model* which consists of a set of features, their relations, and their dependencies. Mapping features to artefacts is not always considered in feature modelling, however it is required if the resulted feature model will be used to provide derivation support. Feature models are typically used to model features belonging to the problem space however they have also been used to represent the solution space (e.g. architecture [Weiler 2003] or source code level [Czarnecki and Eisenecker 2000]). Decision modelling approaches focus on variability modelling and derivation support. Therefore decisions are first class citizens in decision modelling techniques and result in a *decision model* which consists of a set of decisions and their dependencies. Decision models define the problem space variability; product derivation is supported through linking the decisions to the reusable assets of the product line [Schmid et al., 2011]. Mapping decisions to artefacts is an essential aspect of decision modelling approaches [Czarnecki et al., 2012]. Feature modelling approaches originate back to the Feature Oriented Domain Analysis (FODA) method [Kang et al., 1990]; decision modelling approaches originate back to the Synthesis method [Synthesis, 1993].

The first feature modelling technique FODA (Feature Oriented Domain Analysis) was defined in the early 1990's [Kang et al., 1990]. FODA describes how to define characteristics of a certain domain and how to define their commonalities and variations. The feature models in FODA have a graphical notation. FODA's feature models show in a tree-based manner how features relate to each other, either via a composition relation or via a type relation. In this context, Kang defines a feature "*as an increment in the program's functionality*". Since then, features have been a convenient term to refer to system capabilities when modelling variability. A feature is considered as the smallest noticeable building block that adds to functionality in software. Furthermore, features are abstractions that different stakeholders can understand. Naturally, stakeholders speak of product characteristics i.e. in terms of the features the product has or delivers. Furthermore, several extensions for the original feature modelling technique were defined to extend its expressiveness and modelling capabilities. These visual representations are all called *feature models* [Kang et al., 1990] [Van Gurp et al., 2001] [Asikainen et al., 2007] or *feature diagrams* [Schobbens et al., 2007] [Czarnecki & Wasowski, 2007]. Feature models model the variability in software by defining all the possible features that distinct the different products a product line could hold.

As already mentioned, each possible product of the product line encloses a different set of features; this allows creating several distinct products. Defining and dealing with all the different features of the software in order to be able to produce the different products is a challenge due to several reasons: firstly, today's products hold a large number of features with different granularity, and which belong to different stakeholders. Secondly, features do not exist in isolation; rather features interact with one another resulting in a set of dependencies between these features. Those dependencies in addition to the variability restrictions on the features influence the coexistence (or absence) of features in the final product(s). Thirdly, an increase in the number of variable features increases the complexity of deriving member products of the product line. The key issue for success is to have a balance between variability and complexity [Codenie et al., 2009].

² It is also referred to as *variability management* in some of the literature of SPL. Throughout this text, we will use the term variability modelling to refer to the modelling (i.e. identifying and specifying) of variability. Variability management is used in literature of SPL as a much broader term that refers to the process of managing variability through domain engineering phases.

Furthermore, a software product line often undergoes adjustments to meet the continuous changes in customer and market requirements (i.e. evolution process). Keeping this under control at an affordable cost is still a major problem. Increasing the scope and diversity of the products that the product line delivers, results in several serious problems both at the domain analysis level (i.e. modelling level) (see e.g., [Ajila and Kaba, 2008] [Acher et al., 2009]) and at the architecture level (see e.g., [Van Ommering and Bosch, 2002] [Bosch, 2005]). As the product line matures, its scope may significantly widen due to the introduction of new features. Introducing new features propagates from the requirements to the design and then to the implementation. To allow safely adding these newly defined features, dependency relations can be used to anticipate and manage the software product (line) evolution process. For example, some decisions need to be made in order to add or remove certain features in addition to their relations and dependencies.

Furthermore, during the lifecycle of the product line (which is typically longer than that of a single product) there is a need for different types of information, for example information about sources of variability, variable features, dependencies between features, the different stakeholders involved with these features.... etc.. All this information is important for taking adequate decisions for proper variability management and variability realisation. Furthermore, different people are interested in this information for different reasons, and therefore require different abstraction levels and different levels of details. For example, developers need the information in order to understand the knowledge about the different features and how they influence each other (i.e. feature interactions and dependencies) when they have to update software, while the management will need the information to explore which new products can be produced at short notice.

1.2 Research Scope

In this thesis, we are interested in identifying and representing (i.e. modelling) variability at a conceptual design level during the domain analysis phase. This means that we will not consider implementation issues, realisation techniques for variability, or derivation support (i.e. product line configuration) issues. Readers interested in these phases may refer to the systematic literature review on variability realisation techniques by Svahnberg et al. [2005], and the work of Rabiser et al. [2010], which provides a systematic literature review on product derivation support.

The aim of this thesis is to bring variability modelling and variability information management one step closer to companies. Because we are interested in modelling variability for the sake of analysing and understanding it, our research is situated in the area of *Feature Oriented Variability Modelling* as opposed to Decision Oriented Variability Modelling. We believe that characterizing the software in terms of “features” (as done in Feature Oriented Variability Modelling) is a convenient way for modelling variability and commonality, and the term feature can be easily communicated to different stakeholders. Within the context of this thesis variability information management refers to information management of variability and commonality information. Information management of how the variability is designed, implemented, and later instantiated in the application engineering phase is out of the scope of this thesis.

Several works have explored the relation between features and code artefacts (for example, work of Heidenreich et al. [2008], Heymans et al. [2012], and Günther and Sunkle

[2012]). In addition, some commercial tools (e.g. DOORS³, Pure::Variants⁴) allow to create traceability links between features and code artefacts. Although, investigating feature to code relations is interesting and relevant, it is outside of the scope of this thesis. We refer to the chapter on Future Work (chapter 12) for a brief discussion of this issue.

1.3 Problem Statement

By nature software development is a complex task; a great part of the complexity comes from the huge amount of knowledge that needs to be explicitly defined and agreed upon before the actual development process takes start. Additionally, there is the problem of communicating this knowledge to the different people involved in the software development process. Natural language, although very expressive, is also ambiguous and not appropriate for conveying the intended meanings correctly. There exists a lot of knowledge but it is not always related, nor consistent, complete, or accessible. This situation leads to making assumptions about the intended meanings. The difficulty this creates is the continuous growing of the size of these assumptions as we go deeper in to the development of the software, leading to unanticipated results and sometimes failure, because the resulted software is doing something different than expected. This is why conceptual modelling is considered an indispensable step in software engineering.

In the case of introducing variability, this situation becomes even more complicated. Introducing variability to software increases the complexity of the software development process even more. Therefore, it becomes necessary to deal with this additional complexity as early as possible, and therefore also during modelling. The key issue for success is to have a balance between the added flexibility the variability offers, and the complexity the variability brings to the development cycle. To help reaching this balance, during modelling *the involved stakeholders should be able to share their knowledge and understanding of the domain*, i.e. the variable features, the reasons that drove this variability, *and the complexity added by introducing new features should be made explicit*. For example, it is essential to understand which features are variable, how they can vary (i.e. their allowed variations), and which features can, or cannot, or must be combined within products. Furthermore, it is important that the reason for variability does not get lost. *At any point in the product line's lifetime, it should be possible to inspect information on the existing variable features and understand their impact on the overall variability of the system*. A recent study, [Chen and Babar, 2010] about variability management in industrial settings, reports: "how to document variabilities in a way that is easy to understand and use by different stakeholders is an issue". Furthermore, ambiguities in representing variability and commonality information about features may later on lead to misconfigured software, i.e. software that holds incompatible features. It may also lead to lost opportunities, i.e. some possible configurations are not found because they were "thought" as being not possible. *Therefore, ambiguous models should be avoided*.

An additional difficulty in introducing and managing variability comes from the fact that software systems have grown in terms of the number of features they hold and the complexity of relations and dependencies between these features. Feature models can become very large due to this increasing number of features ranging from a few hundred and jumping up to a few thousand [Bosch, 2005]. This makes it difficult to keep a good overview of and maintain the different relations between the different features of the system. *It raises the need to deal with the complexity while allowing efficient identification and management of the*

3 DOORS, <http://www-01.ibm.com/software/awdtools/doors/>

4 Pure::Variants, http://www.pure-systems.com/pure_variants.49.0.html

different features along with their variability and commonality. This calls for firstly, scaling down the complexity of modelling such complex systems. Abstraction mechanisms are in general used to deal with complexity. However, FODA and most subsequent FODA-based feature modelling techniques lack explicit abstraction mechanisms to deal with complexity. Usually, high-level features are decomposed into lower level features in the feature model, but this simple abstraction mechanism does not allow dealing with the complexity introduced by the many different aspects that need to be considered in modern software systems (e.g., hardware aspects, user interface aspects, network aspects, ...etc.). One solution proposed to overcome this issue was the introduction of different categories and the classification of each feature to a certain category [Kang et al., 2002]. However, this categorisation is very fragile and impractical (more details are given in chapter 5). In reality, a feature may have many faces which make categorizing features into a single category a difficult task and therefore not a viable solution for dealing with this kind of complexity.

Secondly, it calls for efficient inspection methods for the created models in order to assist taking decisions. Indeed, different stakeholders would want to inspect the models (mainly inspecting feature dependencies and relations) in order to find out where the complexity comes from, where it can be scaled down, which features are causing an increase in the complexity, etc. Current feature modelling tools have made dealing with large models not any easier [El Dammagh and De Troyer, 2011] [Hubaux et al., 2010a], although some efforts have been done in order to provide better visualization of large feature models [Cawley et al., 2008] [Nestor et al., 2008] [Cawley et al., 2010]. According to Classen et al. [2011], one way out of this is by introducing textual variability modelling languages rather than visual ones in order to overcome the scalability - productivity problems. Yet, while textual modelling languages (e.g. TVL [Classen et al., 2011], Clafer [Bağ et al., 2010]) may indeed improve productivity of software engineers⁵, they lose the cognitive benefits a visual modelling language has to stimulate communication and sharing of ideas between different stakeholders. The problem comes even at the level of communicating features to customers, as reported by Chen and Babar [2010] in their survey for variability management challenges.

Today, the development of software is usually distributed over different teams. As a result, feature modelling will also be distributed. When different teams or persons are involved in the modelling of different parts of the system, the management of the modelled information about the different features is also more difficult. In addition, features are not isolated, and many feature interactions may exist between models developed by different teams. Typically, there are many relations between the features of one single component/subsystem. Moreover, many interactions, dependencies and conflicts may exist between the features of different components. As reported by Chen and Babar [2010] in a recent survey, practitioners have a problem in harvesting and sharing knowledge in their variability models (e.g. feature models). *Therefore there is a need to make this knowledge explicit and readily available for everyone involved in the modelling process and at different moments in time.*

Furthermore, software product lines acquire their variability from the variability in the domain (i.e. problem domain), from the need of different customers, and from software innovation opportunities. Therefore, it is likely that how a feature contributes to the variability of the system changes overtime. For example, in mobile phones five years ago multimedia message sending was an optional feature, not supported by all phones, while today it has become a mandatory feature supported by every new phone. Therefore, *feature modelling should allow (re)using the same feature (or rather feature specifications) with different variability specifications in different contexts or at different points in time.* Furthermore, we

⁵ These textual variability languages fit best software architectures and engineers as pointed out by Hubaux et al., 2011

should not limit this reuse of features during modelling to the concept of a (single) product line. When features are designed well (and specified independent from their current variability contribution) it should be possible to reuse them in different product (lines) belonging to the same or related domains. This is because features are more stable in their core nature than how they contribute to the variability of a system, as this will in general change based on new emerging needs or driven by technology. Therefore, it should be possible to reuse existing fragments of feature models. On the one hand, this could save time; on the other hand, reuse in the domain engineering analysis and design phases could improve reuse possibilities at development time. To the best of our knowledge, the issue of reusing previously defined feature model fragments at the domain analysis stage has not been addressed. We have only encountered works on reusing configuration data, i.e. common patterns in configuration [Behjati et al., 2012], the objective of that work is to facilitate product configuration in case of very large systems, which is not in the scope of this thesis.

1.4 Research Questions

This thesis addresses the problem of modelling commonality and variability of variable software, as well as providing support for the management of this information by the different stakeholders. The problem statement boils down to investigate how to support the variability modelling practice for the current large scale and complex software by addressing current challenges and limitations, on the one hand; and, to provide different stakeholders involved with variable software the necessary support for the storing and querying of commonality and variability modelling information, on the other hand.

Based on the observations made in the previous section, we have formulated the following research questions and related sub questions:

RQ1: How can variability and commonality modelling in today's large and complex systems be supported by addressing current challenges and limitations?

- RQ1.1 Do current feature modelling techniques provide means to understand and express complexity?
- RQ1.2 What are the limitations and practical issues of current mainstream feature modelling techniques? How can we overcome them?
- RQ1.3 What kind of support can be provided during variability and commonality modelling to deal with large and complex systems?
- RQ1.4 What guidelines and support can we provide to stakeholders in identifying features and their variability and commonality?
- RQ1.5 How can the principle of “modelling with reuse” be introduced to feature modelling?

RQ2: How can the knowledge in feature models and features be captured and unlocked?

- RQ2.1 How can the knowledge in feature models and features be captured?

- RQ2.2 How can communication and information sharing between the different stakeholders be supported in order to comprehend and find information concerning the features of the system, their dependencies, and variability?

1.5 Positioning of the Research

In this section we position our research with regard to other scientific work in the context of our research. As already mentioned in section 1.2, our research is situated in the area of Feature Oriented Variability Modelling as opposed to Decision Oriented Variability Modelling.

1.5.1 Feature Modelling Methods

Numerous graphical variability modelling techniques/methods have been proposed and many efforts have been made to classify and compare the different techniques/methods, for example Svahnberg et al. [2005], Sinnema and Deelstra [2007], and Chen et al. [2009]. Yet, an obstacle for their adoption by industry is that the conceptual foundation of the modelling methods is in many cases unclear [Asikainen et al, 2007] [Chen et al., 2009]. Additionally, in many methods the guidelines on how the models should be created is vague. Very little attention has been given to the process of variability modelling itself (i.e. the method).

The fact that the meaning of the modelling concepts is often unclear and no proper guidelines or methods exist, has resulted in the fact that these modelling techniques are not well adopted outside the research community (as reported by Hubaux et al [2010b]). Furthermore, another challenging task for practitioners, as revealed in a recent study by Chen and Babar [2010], is how to harvest and share the information in an efficient way.

Therefore, in this thesis we focus on providing rigorous meaning for the modelling concepts used and providing proper methods and guidelines that practitioners can use to create the models.

1.5.2 Feature Modelling for Large and Complex Systems

One of the purposes of this thesis is to provide means to facilitate variability modelling of *large and complex* software systems. In this thesis we are looking to the complexity of a software system from the viewpoint of understanding and representing the complexity emerging from the large number of features, relations and dependencies. In this context, we refer to the description of Herbert Simon [1981]: *“Roughly, by a complex system I mean one made up of a large number of parts that interact in a non simple way. In such systems, the whole is more than the sum of the parts, not in an ultimate, metaphysical sense, but in the important pragmatic sense that, given the properties of the parts and the laws of their interaction, it is not a trivial matter to infer the properties of the whole. In the face of complexity, an in-principle reductionist may be at the same time a pragmatic holist.”*

To deal with complexity, the principle of Separation of Concerns (SoC) has been used by researchers of both the information systems and the software engineering communities. Also in this thesis, we will use the principle of SoC to deal with the complexity of large systems.

In recent years, several researchers in the Software Product Line community have investigated the SoC principle to deal with the complexity of feature models. For example, Tun

et al. [2009] use the SoC principle to relate requirements to feature configurations, for this purpose three different types of feature models were created. Hubaux et al. [2011] use SoC to provide different stakeholder views or perspectives on large feature models for the sake of facilitating the configuration process. Similarly, Schroeter et al. [2012] use user specific concerns to create different perspectives for configuring large feature models. These works address SoC for the purpose of facilitating configuration by feature models; they do not address the creation of feature models by using SoC. Acher et al. [2012] propose creating fragments of feature models to overcome the large size and complexity of the one feature model paradigm. They also propose operators to merge these fragments. The fragments represent units of focus, no guidelines were proposed for how they are defined; it is entirely up to the modeller to decide on this.

In this thesis, we propose a SoC approach for modelling variability and commonality in large systems. We also present means to deal with the information these large models contain. To the best of our knowledge there is no work on investigating the application of the SoC principles to the conceptual modelling of feature models.

1.5.3 Reuse and Feature Modelling

Reuse⁶ in software engineering often refers to reuse of software artefacts (components, code libraries, templates, etc.) at the code level. In the context of this thesis, we consider reuse in the context of conceptual modelling, i.e. reusing of *features* or parts of *feature models*. *Modelling with reuse* has been explored by several conceptual modelling researchers to facilitate reuse at a conceptual modelling stage. It allows making use of previous knowledge and experiences, which reduces the modelling time for new systems. For example, Welzer et al. [1999] propose the reuse of conceptual models or parts of them for database design; Babenko [2003] proposes the reuse of information in UML models for supporting partial reuse of UML models; Batista et al. [2012] note that different information system projects usually have common behaviour patterns therefore they propose a framework that facilitates reuse of these patterns (using UML model fragments) during requirements engineering.

In this thesis, we focus on supporting feature modelling for *systematic reuse* (i.e. modelling *for reuse* and *with reuse*) rather than having opportunistic reuse (i.e. modelling *with reuse*). Moreover, supporting reuse at feature modelling level should also propagate to the design and development allowing systematic reuse rather than having opportunistic reuse between the different product lines. This is because features should later be mapped to code artefacts. As already mentioned, several works have explored this relation between features and code artefacts (for example, work of Heidenreich et al. [2008], Heymans et al. [2012], and Günther and Sunkle [2012]). Investigating feature to code relations explicitly is however out of the scope of this thesis. Therefore, the focus for reuse is at the modelling level.

1.5.4 Knowledge Management and Software Models

Applying knowledge representation and reasoning (KR), and knowledge management research to the problems of software engineering (SE) has gained a lot of interest both in the software engineering community and in the knowledge engineering community [Alexander Borgida, 2007] [Bjørnson and Dingsøy, 2008]. Many knowledge representation techniques are

⁶ In software engineering *reuse* is the process of implementing or updating software systems using existing software assets. [DOD Software Reuse Initiative, 1996]

used (more details are given in chapter 4), the Web Ontology Language (OWL [OWL Web Ontology Language Overview, 2004]) in particular is favoured because it supports the open world assumption, allowing to reason over incomplete knowledge. The separation of the concepts of *consistency* and *completeness* mean that an evolving model can be checked for consistency, without the incompleteness of the model causing a problem. In contrast, closed world systems make no distinction between incomplete and missing knowledge; any fact not known is assumed to be false [Russell and Norvig 2003].

Many synergies between software models and ontologies have been proposed, for example in the area of reasoning on software models (e.g., Jekjantuk et al. [2011] apply ontology reasoning to diagnose software models), in requirements engineering (e.g., Kossmann et al. [2008] define an ontology driven requirements engineering methodology), in relating feature models to ontologies (e.g., Czarnecki et al [2006] explore the synergy between feature models and ontologies), in representing and validating Model Driven Architectures (MDA) (e.g., Pahl [2005], Walter et al. [2010] propose using ontology encoding and reasoning for MDA models), for agreement on models in large systems (e.g., Oberle et al. [2006] use ontologies to formalise software models), for boosting software comprehension (e.g., Witte et al. [2007] use ontologies to support software maintainers in understanding code allowing querying and DL reasoning support over the code and its documentation).

In this thesis, we use knowledge management techniques and more in particular ontologies to represent the variability and commonality information between features captured by our feature modelling technique (see chapters 6 and 10 for more details). We benefit from the use of this information representation by using the existing reasoning and management support provided for ontologies. However, in addition, we support stakeholders to share and comprehend the feature models by providing them with a dedicated browser that allows them to interactively explore and query these models (see chapter 10 for more details).

1.6 Research Approach and Methodology

To tackle the research questions formulated, we have adopted the Design Science Research Methodology (DSRM) defined by Peffers et al. [2008], which aims applying the design science⁷ approach to fortify the theoretical foundation of research on information systems. The solution presented in this thesis spans several research domains (as shown in figure 1.2), namely software variability modelling and particularly feature modelling which is our problem domain. In addition, to solve our research questions we apply research from knowledge engineering, particularly conceptual modelling, and knowledge representation and reasoning (in chapter 4 we provide some background

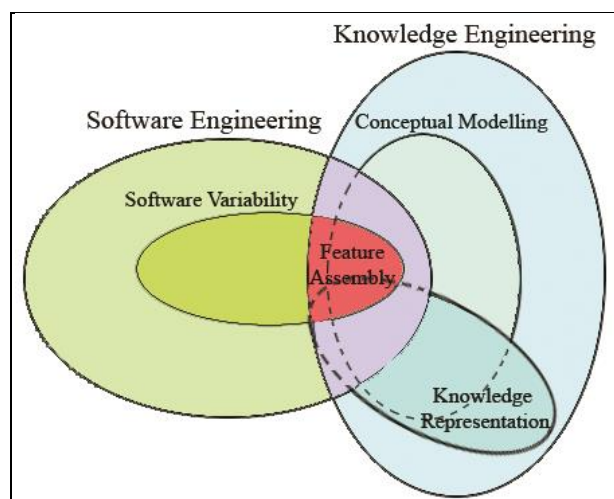


Figure 1.2: Research Areas related to the Feature Assembly Approach Presented in this Thesis

⁷ Design science is fundamentally a problem-solving paradigm whose end goal is to produce an artefact which must be built and then evaluated [Hevner and Chatterjee, 2010].

information related to these domains).

The DSRM process includes six steps: 1) problem identification and motivation, 2) definition of the objectives for a solution, 3) design and development, 4) demonstration, 5) evaluation, 6) communication. The first step, problem identification and motivation have been covered by section 1.3. In the rest of this section, we explain how the other steps have been realized.

1. Definition of the objectives for a solution

In order to define the objectives for the solution, we conducted an empirical literature study for the variability modelling techniques, particularly feature oriented domain analysis techniques because they have the objective of modelling variability and commonality of the system's features. The objective of our study was to determine the most dominant current feature modelling techniques, understand what each of these techniques contributed to the domain, and to identify their limitations in order to identify where improvements could be made. The results of our feature modelling literature study are presented in chapter 2. While doing so, we aimed rationalizing the significance of the problem mentioned in section 1.3 and the appropriateness of the research questions listed in section 1.4 on tackling this problem in order to provide a solution.

Furthermore, the research on modelling and managing software variability is quite diverse and takes several perspectives. The problem statement (mentioned in section 1.3) spans a number of issues that solutions in other perspectives have also tried to address. To distinguish our proposed solution and to highlight how we address the problem differently, we have also considered some of the significant works from these areas of research. This is presented in chapter 3.

This study has set the foundations for our proposed solution which tries to answer the research questions RQ1.1 and RQ1.2 by defining the current limitations and their consequences on the proposed solution (this is presented in chapter 5). Answering these questions helped us gain a better understanding of the problem and therefore define a set of objectives that should be satisfied by the proposed solution. This step serves as input for our proposed solution.

2. Design and development of a solution

Our proposed solution is the Feature Assembly approach, which aims to satisfy the objectives of a solution defined in the first research step.

There are three basic issues involved: Firstly, defining a proper formalism for identifying and modelling knowledge about features, their relations, and dependencies, taking into account the need for scalability, flexibility and feature reuse. The solution we propose for meeting this objective is the *Feature Assembly Modelling technique* (presented in chapters 6 and 7) which answers RQ1 (except RQ1.5) by meeting the objectives for a solution that were identified based on in the previous stage, these objectives are summarized below:

- Support feature modelling with separation of concerns. Identify what could be relevant “concerns” and how they can be defined.
- Provide a method⁸ for feature modelling.
- Provide intuitive, unambiguous, and comprehensive feature modelling concepts and notations.
- Provide modelling concepts to support reuse of features with different variability specifications.

⁸ A method is defined by March and Smith [1995] as a set of steps (an algorithm or guideline) to perform a task.

Secondly, providing support for reusing features during modelling. The solution we propose for meeting this is the *Feature Assembly Reuse Framework* (presented in chapter 9), which is an approach that allows combining both variability and reusability at design time, therefore gaining the merits of both techniques. The provided solution answers RQ1.5 by meeting the objectives for a solution that were identified based on in the previous stage, these objectives are summarized below:

- Define a framework that supports modelling with reuse.
- Define guidelines for modelling with reuse.

Thirdly, providing a managing mechanism for the information about features, their relations, their dependencies, their description, their involved stakeholders, etc., such that this information is readily available whenever there is a need to consult it. The solution we propose for meeting this objective is the *Feature Assembly Knowledge Management Framework* (presented in chapter 10) which answers RQ2 by meeting the objectives for a solution that were identified based on in the previous stage, these objectives are summarized below:

- Support the capturing of information so that it is readily available for the different stakeholders involved.
- Facilitate communication of information and collaboration between the different stakeholders at any point in time through querying, browsing and visualization.

3. Demonstration

In order to demonstrate the usefulness of the presented solution, we have applied it to a Quiz product line case (chapter 8). The presented quiz product line contains 246 features defined in four perspectives, and holds 45 different feature dependencies. By this non-trivial case we demonstrate the modelling of a relatively large and complex system using different perspectives. The example also shows that the modelling notations and semantics are simple to use, expressive, and easy to understand.

We also use the Quiz product line as a running example in subsequent chapters to demonstrate the Feature Assembly Reuse Framework and the Feature Assembly Knowledge Management Framework presented in this thesis.

4. Evaluation

Our evaluation is twofold, firstly, we show that the *Feature Assembly Modelling technique* overcomes the limitations (mentioned in section 5.1) found in mainstream feature modelling techniques (this is done in section 6.6). Secondly, we have tried out the proposed *Feature Assembly approach* in a company to get feedback on the appropriateness of our solution in real settings. This exploratory case study is presented in chapter 11.

5. Communication

We have communicated the solutions defined in this thesis to industry through the VariBru⁹ research project (in which context this research has been carried out) in which we had the opportunity to meet representatives of companies developing software (intensive) systems and discuss their variability challenges and needs. We have also communicated the findings of this thesis in well recognized international conferences and workshops (a list of publications by the author in the context of this thesis is provided on page III).

⁹ www.varibru.be

1.7 Research Contributions

The main contribution of this thesis is the Feature Assembly Approach that deals with the problem descriptions given in section 1.3. This approach consists of a feature modelling technique, a feature reuse framework, and a knowledge management framework. We will discuss the contribution of each below.

1. The Feature Assembly Modelling Technique

As part of the Feature Assembly Modelling Technique, we introduced the *Feature Assembly Modelling Language*, and the Feature Assembly multi-perspective approach. Each brings the following contributions:

• The Feature Assembly Modelling Language

- A new feature modelling language with a few and simple modelling constructs allowing a complete representation of the domain, i.e. the features, their commonality, and variability, in addition to their feature dependencies.
- A language that forces more rigorous modelling by providing a clear separation between *composition* and *generalization-specification* relations. This eases the modelling decisions but also enhances reuse.
- A language that enables reusability of features by separating the specification of the information about the variability from the definition of the features.

• The Feature Assembly Multi-Perspective Approach

- An approach that allows dealing with large and complex software during feature modelling by using the notion of separation of concerns while modelling.
- An approach for defining features that allows abstracting from issues that are not relevant for a particular aspect or viewpoint.
- The use of *perspectives* in feature modelling allows providing dedicated definitions for the concept “feature” for its use in the different perspectives. This provides more guidance to the users than the currently available very open definitions for the concept feature.
- By expressing dependencies between features of different perspectives, the different perspectives are connected with little effort. There is no need for a time consuming integration phase.
- A dedicated perspective, the *persistent perspective*, for dealing with variability in persistent data. We provide a method for deriving this perspective from the other perspectives and for creating the corresponding variable data model.

2. The Feature Assembly Reuse Framework

The Feature Assembly Reuse Framework supports feature reuse during modelling by storing features in a so-called Feature Pool, acting as a feature repository. It brings the following contributions:

- A meta-data based repository that can be searched by modellers for reusable features (possibly created by other modellers for other products).

- A continuously growing repository containing, for a company, all its reusable features (whenever a new feature is defined, it is added to the pool).
- A method for creating feature models to define new variable software by (conceptually) assembling features from the pool with new features, thus supporting creating feature models with reuse.

3. The Feature Assembly Knowledge Management Framework

The Feature Assembly Knowledge Management Framework is a knowledge-based framework that allows representing, and validating feature assembly models. It brings the following contributions:

- An ontology (OWL) format to capture Feature Assembly models.
- A list of SWRL rules that define conflicts or inconsistencies in the models as well as rules that infer information regarding variability.
- A framework that unlocks information captured in feature assembly models, as well as new knowledge inferred by reasoning over the stored information to support finding hidden dependencies, anomalies, and conflicts in very large models.
- A dedicated *Feature Assembly browser* that allows stakeholders to visually explore and interact with Feature Assembly models, as well as with a Feature Pool.
- An OWL representation of the Feature Pool is to support retrieving information about reusable features, applying the same knowledge-based approach as for the Feature Assembly Models.

1.8 Thesis Outline

The rest of the thesis is structured as follows.

- *Chapter 2* describes the background in the domain of software feature modelling. The chapter starts with introducing software product lines and gives an example of a software product line. Next, the term feature modelling is introduced. This is followed by reviewing the different available feature modelling techniques. Our observations on the limitations of feature modelling techniques (Chapter 5) are based on this review.
- *Chapter 3* discusses related works on *representing and analysing feature models, the use of feature models for configuration, modelling with separation of concerns, model integration and consistency checking, variability modelling and databases, and feature model visualization*.
- *Chapter 4* provides some background information related to the domain of knowledge management, as we adopted a knowledge management approach in this thesis. The chapter starts with introducing the importance of conceptual modelling. Next, some of the most widely used knowledge representation techniques are listed. Next, the web ontology language OWL is introduced. Different techniques that support interacting with OWL are also introduced. The chapter concludes with related works in the domain of knowledge management applied to software variability information management.

- *Chapter 5* discusses the different challenges related to software variability modelling. The chapter starts by discussing the limitations encountered in mainstream feature modelling techniques. The second part of the chapter provides the different challenges associated with managing the information contained in variability models. This chapter acts as a knowledge acquisition study intended to identify the requirements for our own approach.
- *Chapter 6* presents the first major contribution of this thesis, the Feature Assembly Modelling Technique, which is a feature oriented variability modelling technique. First, the chapter discusses how a feature can be identified. Next, the multi-perspective approach adopted in Feature Assembly is presented. Then, the Feature Assembly Modelling Language is presented. We conclude the chapter with providing evidence that the Feature Assembly Modelling Technique overcomes the limitations mentioned in section 5.1.
- *Chapter 7* presents the support provided by the Feature Assembly approach for modelling data intensive variable software. The chapter presents the Persistent Perspective, which is the perspective provided in Feature Assembly to define persistent features (i.e. features related to persistent information). Having defined a persistent perspective, the second part of the chapter presents how a link between Feature Assembly Models and Data Models can be achieved.
- *Chapter 8* demonstrates the Feature Assembly Modelling approach with an elaborated example, a Quiz Product line. The Quiz Product line is modelled using the Feature Assembly Modelling approach. Furthermore, the chapter illustrates the flexibility of the presented modelling approach.
- *Chapter 9* presents the second major contribution of this thesis, the Feature Assembly Reuse Framework. The chapter introduces the concept of reusing previously modelled features when modelling new products. The concept of a “Feature Pool” is introduced as a repository of reusable features. The presented approach promotes reuse as early as the design phase therefore aiming to improve the chances of reuse at an architecture and code level.
- *Chapter 10* presents the third major contribution of this thesis, the Feature Assembly Knowledge Management Framework. The presented framework shows how knowledge in feature assembly models can be formally represented via an OWL ontology. Next, the reasoning capabilities of OWL are used to help isolating modelling errors and conflicts. Additionally, different possibilities for retrieving information concealed in the represented models are provided. The chapter is concluded with applying the same techniques to the Feature Pool, in order to browse, visualize, and query the features stored in the Feature Pool.
- *Chapter 11* provides an industrial demonstration for the approach presented in this thesis. In this chapter, we present our experience in adopting the Feature Assembly approach with the company Antidot.
- *Chapter 12* presents the results of this thesis. A summary of the thesis is provided. Possible future work is discussed.

Chapter 2

Variability Modelling Using Feature Models

In this chapter, we give an overview about the state of the art in software variability modelling using feature models. We start by explaining software variability and commonality, and why there is an increasing need to adopt it. Next, we discuss, in general, the feature modelling technique used to model software variability and commonality at a domain analysis stage. Next, we give an overview of the mainstream FODA-based feature modelling techniques discussing the differences between these techniques in syntax, or semantics, or both. We also give an overview of the techniques that extend UML for variability modelling.

2.1 Software Variability

Over the last years software production has been leveraged in terms of the complexity and size. While complexity and size are growing leading to a longer production time, the turnover¹⁰ time of software is decreasing resulting in more demand for new and more advanced capabilities and causing productivity/profit challenge for software companies. Driven by customers that are increasingly cost-conscious and demanding, more and more companies compete on the basis of “*giving customers exactly what they need*”. More and more customers require specifically tailored products that better meet their needs. Meanwhile, software is being recognized as a powerful tool for differentiation and innovation. This has increased the interest in techniques capable to deliver software that can easily be varied to meet the different needs of different customers. At the same time, there is a need to deliver products rapidly in order to decrease their time to market¹¹ (TTM). In order to support the development of software that can easily be varied, the concept of *software variability* was introduced. Introducing variability in software allows varying some of the software capabilities and functionalities to meet the requirements of different users. Companies have often used techniques such as software customization (e.g., via configuration files that set some of the application parameters), changing and editing existing code, using different code libraries and so on, to vary their software. However, if providing many variants of a software product is done in such an ad-hoc manner (e.g., via continuous customization of the existing code base), the variants of one product could become very diverse making it too complex to keep track of all the produced different variants. As an answer to this, the Software Product Line (SPL) [Bosch, 2000] (also called Software Product Family [Asikainen, 2004]) approach was introduced to allow for rapid development of software that could easily be configured to meet the different requirements of

¹⁰ The turnover time of software refers to the lifetime of the software. Currently the advance of hardware has motivated a demanding need for a similar advance in software. This has resulted in a shorter lifetime for software products and an increased demand for new more advanced software capabilities.

¹¹ Time to market (TTM) is the length of time it takes from a product being conceived until it's being available for sale.

the various customers. The idea of product lines is not new; it has long been applied in industry ranging from car manufacturing and home appliances (refrigerators, washing machines, etc.), to consumer products.

Software Product Lines apply the concept of *product lines* defined in manufacturing to the software development process. It moves the software development from a product-based development to a product line-based development, in which multiple related products are considered from the very beginning of the software development process. The product line can be configured to produce different products meeting the needs of different customers. This is achieved by introducing variability at an early stage in the planning and development of the product line. Therefore, it allows for design and development of a set of closely related software products rather than a single product. This enables efficient reuse of assets¹² during the development cycle, which in return will enable rapid development of related software products. Therefore enabling better productivity, which is the main benefit of applying the product line technique. A software product line is commonly defined to consist of a common architecture, and a set of reusable assets. Together they are used in producing individual products by using a different set of assets in each individual product. A software product line is characterized in terms of its capabilities and characteristics. These are often referred to as *features*. For example an E-Shop product line would have the features *Shopping Basket*, *Purchase*, and *Payment*; while a mobile phone product line would have the features *Call*, *Accept Call*, *Text Message*, and *Multimedia Message*.

Software product lines have gained a lot of attention from the industry due to their ability to:

- **Reduce the time to market:** software product lines allows to rapidly create a family of products rather than one product only, thus improving significantly the development time of a new product because of efficient reuse of software assets.
- **Increased bandwidth to pursue more markets:** the development of a family of products rather than one product allows companies to add different flavours to their products to suit different markets and/or different categories of users. Therefore generating more revenue and be competitive in new markets.
- **Decrease the development time of products:** in software product lines the reuse of assets is planned beforehand therefore increasing the reuse opportunities between the different members of the product line and decrease the time for development and maintainability.

There are several reports from industry on the added value for their organization from adopting the product line approach, for example in the area of *mobile phones* [Maccari and Heie, 2005], *Car Periphery Systems* [MacGregor, 2002], *MRI systems* (Magnetic Resonance Imaging scanners) [Jaring et al., 2004], *web browsers* [van Gurp, 2001], *printer software* [Svahnberg and Bosch, 1999], *eHealth systems* [Bartholdt et al., 2008], *Revenue Acquisition Management solutions* [Clements and Northrop, 2002], and *Web portals* [Pettersson and Jarzabek, 2005].

As an example of the benefits of adopting a software product line approach, consider a company developing *Quiz* systems and having different customers in somewhat different domains. The company wants to deliver to each customer the *Quiz* system that best meets his/her needs. Additionally, they want to make the best available reuse of the existing assets and reduce their development time. For example, *customer A* is a primary school that requires a

¹² Software assets refer to all the artefacts that make up certain software; some artefacts maybe external while others may be internal.

Quiz application for efficiently examining their students. While *customer B*, a higher education institute, needs a more sophisticated version of the application that could both handle simple quizzes and exams to be used by students for online examinations and quizzes. The third customer, *customer C*, is an organization that would like to provide assessments for their new employees allowing them to interactively test their knowledge of the organizations values and processes. The fourth customer, *customer D*, is a company that would like an application which allows them to customize their online marketing surveys. Adopting a product line approach, the software company could recognize some similarity in the required four products. They could define a Quiz Application product line that could be tuned to deliver the above-mentioned products (i.e. applications). While all products will contain the same kernel, each product will contain a set of different features so that each product satisfies the needs of its customer. Even more, such a product line will allow them to serve more potential customers with similar requirements. A sample of the four different products of a Quiz product line is shown in figure 2.1.



Figure 2.1 Sample of Quiz Product Line possible products

As already explained, the goal of software product lines is to plan for the development of a set of closely related software products rather than for a single product. This enables efficient reuse of assets during the development cycle. However, on the other hand, adopting a software product line technique will increase the complexity of the software development process. Identifying and managing the different features of the software in order to be able to produce the different products is a non-trivial task as features are usually not independent. This calls for methods and techniques to deal with the complexity of introducing variability within a product line. The key issue for success is to have a balance between the added flexibility the variability introduces, and the complexity this variability brings to the development cycle.

Software product lines are realized via introducing *software variability* at an early state of the development life cycle. Software variability is defined as “*the ability of a software system or artefact to be efficiently extended, changed, customized or configured for use in a particular context*” [Svahnberg et al., 2005]. In software product lines, variability opportunities are defined at the *domain analysis* phase. Domain analysis is defined by Neighbors [1984] as “*the activity of identifying objects and operations of a class of similar systems in a particular problem domain*”. Domain analysis is also known to be “*The analysis of systems within a domain to discover commonalities and differences among them*”. The output of the domain analysis stage is a definition of a domain model that characterizes the domain (i.e. the product line capabilities). Variability opportunities are indicated in terms of *variation points* and *variants*. Variation points denote the software characteristics at which variability opportunities exist. Variation points are defined as “*places in the design or implementation that together provide the mechanisms necessary to make a software feature variable*” [Svahnberg et al., 2005]. Variation points are introduced due to market requirements, stakeholder recommendations, customer requirements, innovation opportunities, or to increase business opportunities. Variants denote the specific possibilities a feature may have. Generally a variation point could be associated with any number of variants. Variation points are associated with a *binding time*, which denotes the time in the development cycle that a certain variation point will be bound to a specific variant(s). A variation point may be bound to a specific

variant(s) during the product architecture derivation, during compilation, during linking, or at runtime [Svahnberg et al., 2005]. When a product is derived from the product line, for each variation point only a subset of the variants are selected.

2.2 Software Variability Modelling

The software development process is by nature a complex process. Introducing variability to software adds an additional level of complexity to the software development process. To help cope with this complexity, there is a need for efficient variability modelling techniques capable of modelling and documenting variability and commonality at an early stage in the development process. In the context of software product lines, a variability modelling technique should be expressive enough and easy enough to capture and represent information about features composing the software product line, in addition to how these features contribute to the variability of the software product line. The variability model should express the product line capabilities by allowing the representation of commonality and variability within the features. This makes it possible to clearly anticipate allowable feature combinations. In addition, it allows anticipating variability opportunities that might have been implicit or not identified before.

Failing to properly model variability may lead to incorrect and usually difficult to debug software. Furthermore, possible variability opportunities could be missed which means missed business opportunities. For example, conflicting features not anticipated at modelling time are more expensive to solve at a later stage in the development process. Furthermore, a balance has to be made between variability and complexity. Modelling variability provides a better understanding of the available variability possibilities and thus helps making better decisions on which variability opportunities should actually be realized (i.e. actually implemented in the final products) and which ones should be delayed or even ignored. Not all variation points are necessarily realized in the final products, some may have more impact than others. Furthermore, some variation points may be neglected due to their complexity or market immaturity. Similarly, not all variants are of the same importance, a variation point could be realized but only a subset of its variants is realized. Some variants may be of more importance than others. In addition, modelling software variability is of great importance in order to manage the commonalities and differences between the different variants of the product line at an early stage. This allows defining an appropriate architecture and a reuse methodology that best realizes the variability of the defined software product line. Variability models also help domain engineers, project managers and architecture engineers in making decisions about when to bind the variation points to specific variants. The appropriate binding time is influenced by the amount of variability that will actually be realized and the variability realisation technique that will be used.

In addition, variability models¹³ abstract from how the variability will be implemented. This makes it easy to communicate the variability of the software product line to the different stakeholders involved at an early stage of the development process. A variability model can act as a base for communication between the different stakeholders, which usually have different interests and requirements, sometimes even conflicting ones.

The phase in software development in which variability is analysed and variability models are created is called the *domain analysis*. Kang et al. [1990] consider the domain

¹³ In this thesis we stick to feature models. Architectural and realization variability models are out of the scope of this thesis.

analysis process as a factor that can improve the software development process and promote software reuse by providing a means of communication and a common understanding of the domain. The authors define the domain analysis process as “*the process of identifying, collecting, organizing, and representing the relevant information in a domain based on the study of existing systems and their development histories, knowledge captured from domain experts, underlying theory, and emerging technology within the domain*” [Kang et al., 1990]. In case of variable software, domain analysis supports software design by providing a better understanding of the commonality and variability that a certain domain holds. This allows promoting feature reuse over a certain domain, by capturing domain expertise; domain analysis can also support communication, tool development, and software specification and design [Kang et al., 1990].

As already mentioned in the introduction of this thesis, variability modelling techniques are usually based on *feature modelling* or *decision modelling* [Czarnecki et al., 2012]. In Decision modelling approaches decisions are first class citizens in the decision modelling techniques and the result is a *decision model* which consists of a set of *decisions* and their *dependencies*. Decision modelling approaches (e.g. Schmid and John [2004], KobrA [Atkinson et al., 2000], and DOPLER [Dhungana et al., 2010]) focuses on variability modelling and derivation support; therefore how decisions relate to the solution artefacts is explicitly modelled [Schmid et al., 2011]. On the other hand, Feature modelling approaches focus on commonality and variability modelling. Therefore features are first class citizens in the feature modelling techniques, and the result is a *feature model* which consists of a set of *features*, their *relations*, and their *dependencies*. Feature models are typically used to model features belonging to the problem space; however they are also used to represent features belonging to the solution space. In the next sections, we will discuss feature models in general and then give an overview of the characteristics and underlying concepts for the most commonly used ones.

2.3 Feature Models

Careful planning for variability and commonality is a key factor for successfully gaining the merits of using software product lines. By careful planning of variability and commonality we mean clearly defining and representing this information in an unambiguous and well defined form at an early stage (i.e. within the domain analysis phase), this process is referred to as *Feature modelling*. Feature modelling is the process of identifying and representing the characteristics and capabilities of the product line, the output of this process is referred to as the *feature model*.

The first feature modelling technique defined was the Feature Oriented Domain Analysis (FODA) technique defined by Kang [1990], which was intended as a method for domain analysis and modelling, and since then it has become an appealing technique to the software research community for modelling variability in software. Although other domain analysis techniques existed (e.g., STARS [Creps and Simos, 1992], and DSSA [Tracz et al., 1993]), feature oriented domain analysis (FODA) was quickly adopted to effectively identify and characterize the software product line capabilities and functionalities at an early stage. FODA became used for the analysis of variable software due to its ability to represent and model commonality and variability among applications of a certain domain. Each member of a product line (i.e. product) is built up of a specific set of features which identify its capabilities. Furthermore, FODA was applied to several case studies [Kang et al., 2002] and many extensions to the original technique have been defined to extend the expressivity of FODA in order to better meet the needs of modelling variability in software (more details in section 2.4.1).

FODA was intended to capture all the information in a domain in order to capture and document domain knowledge. The power of using FODA is its ability to make knowledge about a certain domain explicit and no longer in the heads of domain experts only. Feature oriented domain analysis was used to identify where the applications for a certain domain are similar and where they vary. Therefore FODA provides constructs that capture variability and commonality within a certain domain. FODA captures the possible applications of a certain domain abstracting from functionality or processes within these applications. This allowed applying FODA to represent software variability. In FODA, applications in the domain are described in terms of *features*. Features are abstractions that different stakeholders can understand. Stakeholders usually speak of product characteristics i.e. in terms of the features the product has or delivers [Kang et al., 2002]. Features are actually user-visible aspects or characteristics of the domain [Kang et al., 1990].

In the context of software product lines, a software feature is commonly defined as an increment in the program's functionality [Batory, 2005]. A feature is considered the smallest building block that adds to functionality of the product line, whether this functionality is external (i.e. visible to users) or internal (operational and not directly visible to users). Furthermore, features indicate capabilities of the system; these capabilities fulfil both the functional and non-functional requirements of the software. Features can differ in their complexity and size. Some features may be fairly simple such as *colour*, *shape*, *language*, or *text direction*, while others may be more complex such as *spelling check*, *shopping cart*, or *purchase*. Different stakeholders may be interested in different features of the system and at different level of details.

FODA represents the domain in terms of visual *feature models* or *feature diagrams*. Feature models relate features by means of a hierarchical tree structure, describing how features are broken up into corresponding constructing (sub) features, with exactly one root node. Features at the top of the hierarchy represent coarse-grained domain concepts while features at the bottom of the hierarchy represent finer grained characteristics of the domain (and later on the application). Feature Models also show how the features contribute to variability. Feature models [Kang et al., 1990] [Van Gurp et al., 2001] model the variability in software by defining all the possible features which different products of a product line could hold.

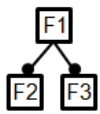
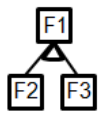
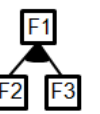
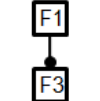
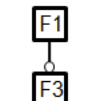
A feature model not only shows the feature composition hierarchy but also shows the relation of the feature with regard to their break up group. The link between a feature (source) and its sibling feature (destination) is called a *feature relation*. Features could have a *mandatory* feature relation, which identifies a compulsory whole-part composition relationship (i.e. this relation holds for all valid products, thus the destination feature should be part of all valid products). Features could also have an *optional* feature relation, which identifies a *voluntary* whole-part composition relationship (i.e. the relation could hold in any valid product, thus the destination feature could be part of any valid product). FODA captures variability in the domain by means of *voluntary* whole-part relations and *XOR* feature relations. *XOR* feature relations define the opportunity to select one feature from a group of features. Furthermore, In FODA two feature dependencies were defined: *Requires* and *Excludes*. Such dependencies describe how features interact with each other and control which features could exist and co-exist together in the same product.

Feature models define the whole spectrum of possible products. How the features are related in the feature model restricts the products that could be derived out of the product line. A feature model defines the set of possible *configurations* of a certain product line. A configuration is defined as a valid composition of features; a valid composition of features results in a valid product, which is a product that meets all the restrictions specified in the feature model.

Typically feature models hold three types of information: Features, Feature relations, and Feature dependencies

- Features:** Features are represented as nodes in the feature model. Features are associated with a *feature type*, which indicates the type of relation they participate in. In general, a feature can have both a group feature type (i.e. OR, AND, or Alternative) and a single feature type such as optional or mandatory. Having more than one feature type should be avoided as it leads to ambiguity and calls for normalizing the feature model (feature model normalization will be discussed below, in section 2.3.1). As an example, a feature model could indicate (by means of AND relations) that for a certain feature all sub-features must be part of any product. An OR feature indicates that it is part of an OR group which holds an OR relationship between its members. Commonly, there are five possible *feature types* in a feature model [Kang et al., 1990] [Batory, 2005], which correspond to five possible *feature relations*; table 1 shows their graphical notation and meaning in terms of a *configuration* [Bosch, 2000] [Van Gurp et al., 2001].

Table 2.1. Graphical Notation of Feature Types and Their Relations, modified after [Batory, 2005]

(a)	And indicates that any configuration that contains the parent feature must contain all the sub-features (i.e. in any configuration: if F1 is selected then F2 and F3 should also be selected).	
(b)	Alternative indicates that any configuration that contains the parent feature must contain exactly one of the sub-features, (i.e. in any configuration: if F1 is selected then F2 or F3 should also be selected).	
(c)	Or indicates that any configuration that contains the parent feature may contain one or more of the sub-features, (i.e. in any configuration: if F1 is selected then F2 and /or F3 should be selected)	
(d)	Mandatory indicates that any configuration that contains the parent feature must contain the specified sub-feature (i.e. in any configuration: if F1 is selected then F2 should be selected)	
(e)	Optional indicates that any configuration that contains the parent feature may or may not contain the sub-feature, (i.e. in any configuration: if F1 is selected then F3 may or may not be selected)	

- Feature relations:** Feature relations represent the branches in the feature model. Feature relations denote a decomposition of features; the coarse grained characteristics are at the top of the feature model tree while their fine-grained decompositions are at the bottom of the tree. The leaf features indicate that no more decomposition is possible (i.e. it adds no information in terms of variability and commonality of the product line features). There are two types of relations, *group relations* and *single relations*, a feature may combine both group relations and single relations. Group relations includes grouping of related features which hold a generalization- specification relationship with their parent feature. In terms of configuration a selection is made based on the type of the group. Three groups of possible feature relations exist; *AND group*, in which all the features belonging to this group should be selected in the final configuration, *OR group* in which some of the features belonging to this group are selected in the final configurations, and an alternative group in which only one of its member features gets to be selected in the final configuration. Single relations on the other hand denote a

simple whole-part decomposition, in the case of mandatory relations the decomposition is compulsory, while in the case of optional relations the decomposition is optional.

- **Feature dependencies:** The *Requires* and *Excludes* defined by FODA represent additional constraints that control which features could exist and co-exist together in a valid configuration. They could be visually represented in the feature model causing it to become a directed graph (DG) or simply added as textual constraints in addition to the tree representation of feature model. These feature dependencies are the ones commonly used by successive feature modelling techniques.

Table 2.1 shows the graphical notation of the different feature types of FODA. The features belonging to an *And* feature group and the features of a *Mandatory* type define the features that are common to all members of the product line. Variability on the other hand is represented by means of the *Alternative* feature group (which represents an XOR relationship between the member of the group), the *OR* feature group (which represents a voluntary relationship between the members of the group), and the *Optional* relation (which represents a voluntary relationship).

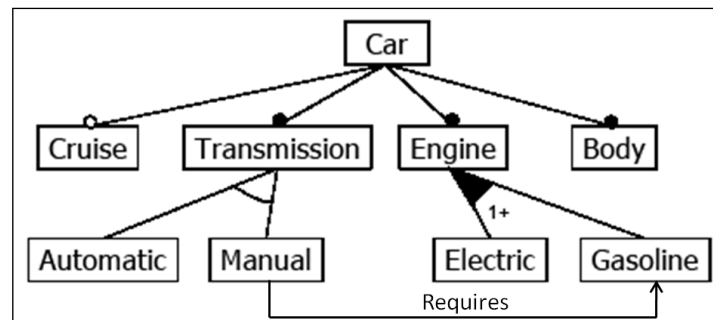


Figure 2.2: Feature Model of *Car* Product Line

Figure 2.2 shows a sample feature model for a *Car* product line. The model represents the following information: a car must contain the following features: a *Body*, an *Engine*, and *Transmission*. It may optionally contain a *Cruise*. The engine could either be *Electric* or *Gasoline*; a car may have both. The transmission of a car should either be *Manual* or *Automatic*; only one should be selected, a car could not have both. Additionally there is a *Requires* dependency between the *Manual* feature and the *Gasoline* features, i.e. whenever the transmission is manual then a *Gasoline* engine should also be selected. The feature model in figure 2.2 suggests that 10 possible variants of a car product can be derived. Listing 2.1 shows the possible configurations for the *Car* product line. The process of deriving possible valid products from a product line is called the *product configuration*; each possible product is called a feasible configuration of the product line.

```

Car1= Cruise + Transmission {Automatic} + Engine {Electric} + Body
Car2= Cruise + Transmission {Automatic} + Engine {Gasoline} + Body
Car3= Cruise + Transmission {Manual} + Engine {Gasoline} + Body
Car4= Cruise + Transmission {Automatic} + Engine {Electric, Gasoline} +
Body
Car5= Cruise + Transmission {Manual} + Engine {Electric, Gasoline} + Body
Car6= Transmission {Automatic} + Engine {Electric} + Body
Car7= Transmission {Automatic} + Engine {Gasoline} + Body
Car8= Transmission {Manual} + Engine {Gasoline} + Body

```



```

Car9= Transmission {Automatic} + Engine {Electric, Gasoline} + Body
Car10= Transmission {Manual} + Engine {Electric, Gasoline} + Body
    
```

Listing 2.1: Possible configurations of Car product line shown in figure 2.2

As illustrated by the example, feature models do not only act as a representation and documentation of the variability and commonality in the system, but they also provide the possible solution space for the set of possible products that could be derived from the modelled software product line. Failing to correctly model the features or correctly indicating how they relate to variability (feature relations) and relate to one another (feature dependencies) results in wrong products or products that do not anticipate the capabilities of the product line. This is because, finding possible configurations is strictly speaking, a constraint-solving problem in which a solution (configuration) is found that satisfies the relations and rules defined in the feature model. A feasible feature model is one that holds no (logical) contradictions or conflicts within the different dependency constraints between features.

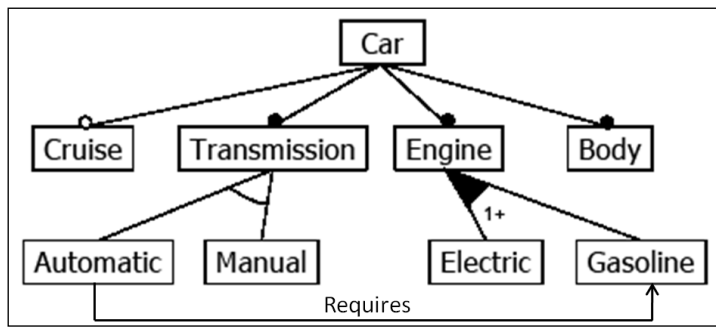


Figure 2.3: Feature Model of Car Product Line

It should be noted that changing any aspect in the model means also a different set of possible products. For instance, in the example, a change in the *Requires* dependency to state that, *Automatic* transmission requires *Gasoline* engines would give a different set of possible car products (see figure 2.3). Listing 2 shows the new set of possible products.

```

Car'1= Cruise + Transmission {Automatic} + Engine {Gasoline} + Body
Car'2= Cruise + Transmission {Manual} + Engine {Gasoline} + Body
Car'3= Cruise + Transmission {Manual} + Engine {Electric} + Body
Car'4= Cruise + Transmission {Automatic}+ Engine {Electric, Gasoline} + Body
Car'5= Cruise + Transmission {Manual}+ Engine {Electric, Gasoline} + Body
Car'6= Transmission {Automatic} + Engine {Gasoline} + Body
Car'7= Transmission {Manual} + Engine {Gasoline} + Body
Car'8= Transmission {Manual} + Engine {Electric} + Body
Car'9= Transmission {Automatic} + Engine {Electric, Gasoline} + Body
Car'10= Transmission {Manual}+ Engine {Electric, Gasoline} + Body
    
```

Listing 2.2: Possible configurations of Car product line shown in figure 2

The result set shown in listing 2 shows that three configurations (Car'₁, Car'₃, and Car'₈) are new.

The above example clearly shows the importance of modelling variable software at domain analysis level. Establishing a model to express variability and commonality ensures a

better understanding of the capabilities of the product line. Furthermore it acts as a formal documentation of these capabilities within the lifetime of the product line. In addition, verifying the correctness of the established models at domain analysis time helps preventing errors at a later phase of the software product line development. Of course, the significance of an error varies according to the problem being modelled and the effect of the error itself on the resulted model. Nevertheless, for reliable and robust variable software, inconsistent and conflicting feature models should be avoided, as they will lead to the creation of incorrect and usually difficult to debug software (incorrect combinations of features could be made). In addition, possible products could be missed which means missed business opportunities.

2.3.1 Normalizing Feature Models

It should be noted that, in FODA (and subsequent feature modelling techniques), combining more than one type of features in a single relation (e.g., an *Or* relation containing *mandatory* siblings) is not prohibited. Yet it increases the complexity of the model and leads to redundant relations. Feature models without redundancy (i.e. each branch only contains a single type of features) are called *Normalized Feature Models* [Czarnecki, and Eisenecker, 2000] [von der Massen and Lichter, 2004].

Normalization is defined as transforming combinations of child features with different types of variability to child features with a single type of variability. As an example, figure 2.4 shows possible normalizations for alternative features. Figure 2.4.a shows how an *optional alternative* feature (i.e. gives the possibility to select zero or one of the alternative features) is reduced to an alternative feature with an optional parent feature. Figure 2.4.b shows how the mandatory alternative (i.e. gives the possibility to select exactly one of the alternative features) is reduced to an alternative feature with a mandatory parent. Similarly, a combination of an *optional OR* could be used to represent the possibility to select zero or more of the *OR* features. While a mandatory *OR* could be used to indicate the possibility to select one or more of the *OR* features. Figure 2.5 shows possible normalization by using the cardinality based feature models described in section 2.4.1, in which each OR group composition is associated with a minimum and maximum cardinality stating the minimum and maximum number of sibling features that are allowed to be selected in a valid configuration

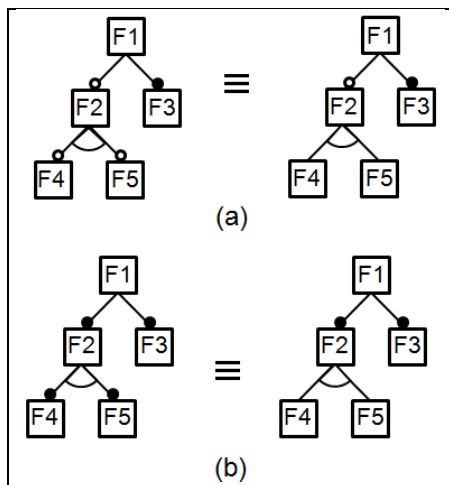


Figure 2.4 Possible normalization for
a) optional b) mandatory alternative features

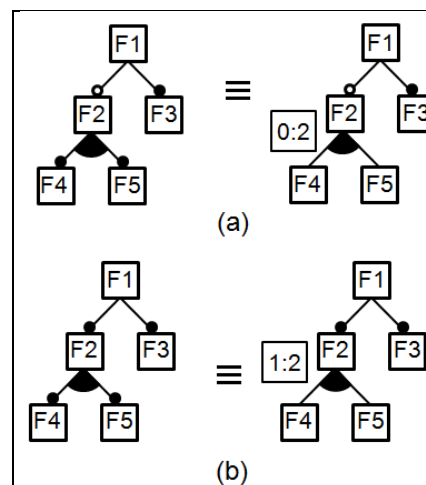


Figure 2.5 Possible normalization for
a) optional b) mandatory OR features

2.4 Mainstream Feature Modelling Techniques

Over the past two decades, several feature modelling techniques have been developed that aim supporting variability representation and modelling. Several extensions to FODA (the first feature modelling language) have been defined to compensate for some of its ambiguities, to introduce easier to use modelling concepts, or to introduce new concepts and semantics to extend FODA's expressive power. They all keep the hierarchical structure originally used in FODA, accompanied with using some different notations.

In order to answer our first research question (RQ1; How can variability and commonality modelling in today's large and complex systems be supported by addressing current challenges and limitations?) we have first studied existing feature modelling techniques and identified their characteristics, in order to identify and analyse their limitations (which serve as input for our proposed solution, this will be discussed in chapter 5).

We describe here the most common FODA based feature modelling techniques (FORM [Kang et al., 1998], FeatureRSEB [Griss et al., 1998], [Van Gorp et al., 2001], [Riebisch et al., 2002], PLUS [Eriksson et al., 2005], and Cardinality Based Feature Models [Czarnecki and Kim, 2005]), which provide a broad overview of the characteristics of mainstream feature models.

2.4.1 Feature-Oriented Reuse Method (FORM)

Feature-Oriented Reuse Method (FORM) [Kang et al., 1998] extends FODA by adding a domain architecture level which enables identifying reusable components. It starts with an analysis of commonality and variability among applications in a particular domain and identifies features of these applications. Features are classified in terms of four different categories (also called layers): *capabilities*, *domain technologies*, *implementation techniques*, and *operating environments*. Capabilities are user visible characteristics that can be identified as distinct services provided by the application (e.g., call forwarding in the telephony domain), operations that the application performs (e.g., dialling in the telephony domain), and non-functional characteristics (e.g., performance) that affect the feature selection process. On the other hand, domain technologies (e.g., navigation methods in the avionics domain) represent the way of implementing services or operations within the application domain. Implementation techniques (e.g., synchronization mechanisms in the telephony domain) are generic functions or techniques that are used to implement services, operations, and domain functions, these techniques can be shared by more than one domain. Operating environments (e.g., operating systems) represents environments in which applications are used.

Common features among different products are modelled as mandatory features, while different features among them may be *optional*, or *alternative* features. Alternative relations are still supported as with the original FODA. A feature model is created with AND/OR nodes, the feature model shows the classification of the features based on the previously mentioned categories. The feature model explicitly represents three types of relations: *composition*, *generalization/specification* and *implemented by*. While the first two relations were originally introduced in FODA, the new *implemented by* dependency was introduced to relate features in terms their functionality, and later on implementation at an architecture level. Figure 2.6 shows a sample feature model using the notation and semantics defined in FORM.

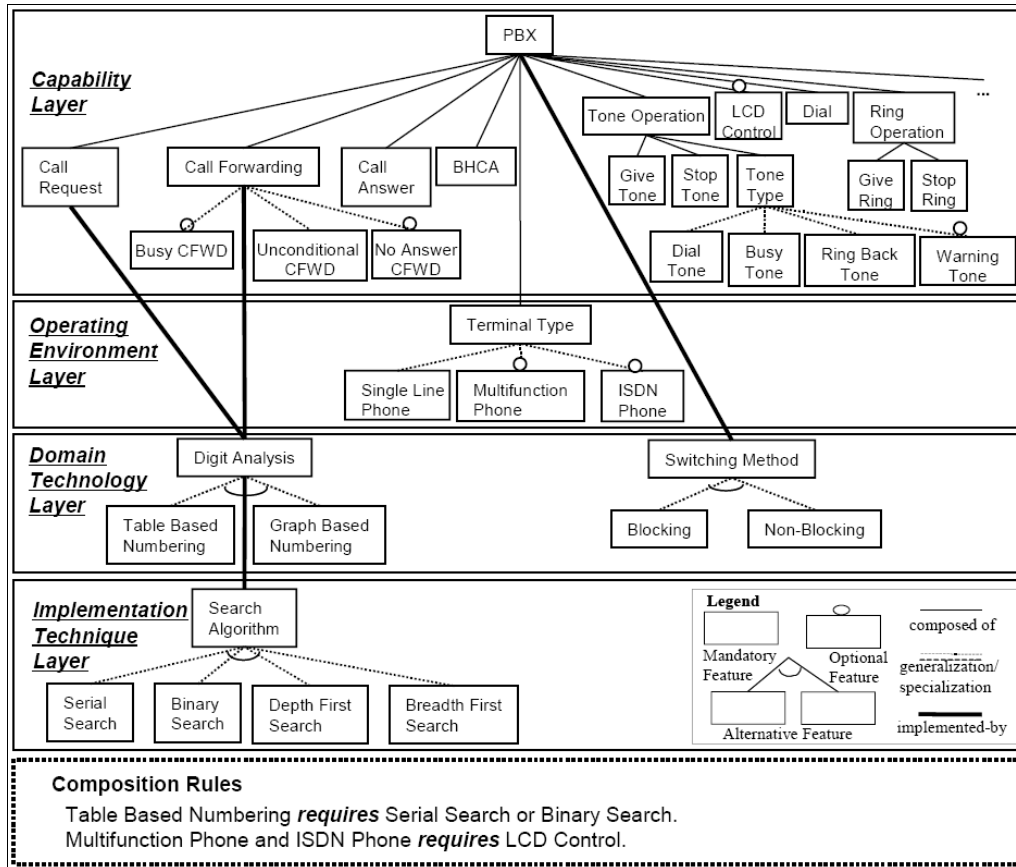


Figure 2. 6 A feature model in FORM for the Private Branch Exchange (PBX) product line, [Kang et al., 1998]

In addition, because it becomes quite complex to link features belonging to different categories, a textual specification language, next to the hierarchical structure, was used to characterize the system. With respect to feature dependencies, the *excludes* and *requires* dependencies originally defined in FODA are still used.

2.4.2 FeatureRSEB

FeatureRSEB [Griss et al., 1998] aims at integrating feature modelling with the Reuse-Driven Software Engineering Business (RSEB) [Jacobson et al., 1997]. RSEB is a systematic, model-driven approach to large scale software reuse, applied to an organization engaged in building sets of related applications from sets of reusable components. In RSEB explicit use case models are central to all steps that define architecture, subsystems and reusable objects. Therefore, FeatureRSEB uses UML use case diagrams as a starting point for defining features and their variability and commonality. The FeatureRSEB feature models created are based on the functionality provided by the many use cases that represent the different possible user requirements for applications of a certain domain. This rationale was based on the fact that UML use cases are presumed to get a better understanding of the user requirements within a certain application domain, while feature models define how in one domain, these applications may differ based on the variability that can be imposed by the different *possible* end product features. Furthermore, feature models are capable of defining the selection mechanism for the final features within any product variant.

In FeatureRSEB, feature models classify features to *optional*, *mandatory* (similar to FODA) and *variant*. A *variant* feature is used to indicate alternative features and OR features, i.e. it represents any set of features in which selectivity is allowed. The OR selection option was introduced to represent the *selection of some options from many* relation. A distinction is made between the two in terms of the notation used, a filled diamond indicates OR selectivity while a hollow diamond indicates Alternative selectivity. Additionally, the concept of variation points was added as part of the model, variation point features are known as *vp-features*.

Branches in FeatureRSEB indicate composition/decomposition relations between features. The *excludes* and *requires* dependencies originally defined in FODA are used to represent constraints between features. They are modelled as separate constraints with respect to the diagram. Figure 2.7 shows a sample FeatureRSEB feature model for Rapid Telephone Service Creation product line.

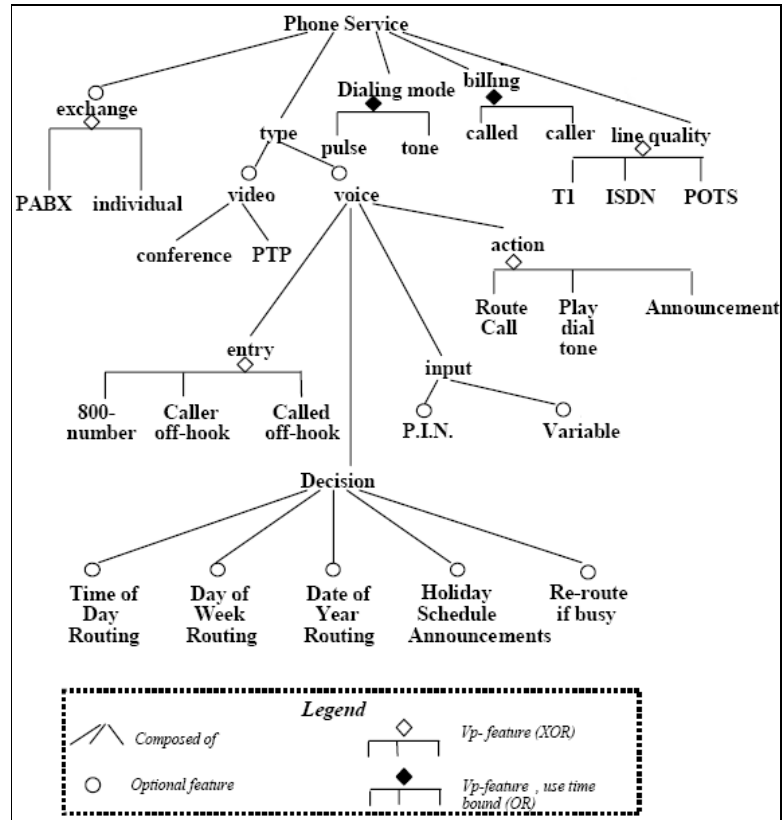


Figure 2.7 A feature model in FeatureRSEB for Rapid Telephone Service Creation product line, [Griss et al., 1998]

2.4.3 van Gorp et al. Feature Graph

Van Gorp et al. [2001] define a new feature modelling method based on FeatureRSEB. They refer to the resulted feature models as *feature graphs*. The authors consider a feature as a construct that should group related requirements. Features in this method are defined as “*a logical unit of behaviour that is specified by a set of functional and quality requirements*”. They use the same feature types as proposed by FeatureRSEB, which are *mandatory* feature, *optional* features and *variant* features. Variant features are either OR features or XOR features. In addition they propose a new feature type named *external feature*. External features are features offered by the target platform of the system. While not directly part of the system, they are important because the system uses them and depends on them. Therefore they have found them to be of importance at the configuration phase.

Additionally, the authors added the notion of *binding time* to their feature graphs. The binding time information indicates the time in the development process that a variation point will be bound to a specific variant. The authors classify variation points to *open variation points* and *closed variation points*. In an open variation point, new variants may be added to the set of variants associated with the variation point. In a closed variation point no new variants can be added once the variants for the variation point are defined. Figure 2.8 shows a sample feature graph for a mail client product line. The *excludes* and *requires* feature dependencies originally defined by FODA are still used to denote restrictions between features.

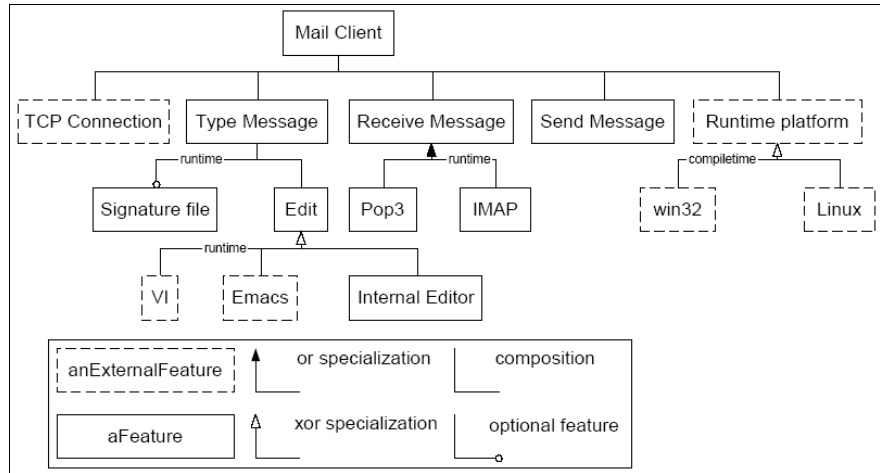


Figure 2.8 van Gorp et al. feature graph for a mail client product line, [Van Gorp et al. , 2001]

2.4.4 Riebisch et al. Feature Models

In 2002, Riebisch et al. proposed to add multiplicity to feature groups to indicate the number of features that are allowed to be selected from each branch. This need comes from the ambiguity of existing feature modelling techniques when it comes to selection of features from within a group. For example, *OR* groups indicate the selection of some features from many but the exact number of allowed features remains unexpressed. Riebisch et al. propose that a set has a multiplicity that denotes the minimum and maximum number of features to be chosen from the set. Possible multiplicities are: 0..1, 1, 0..n, 1..n, m..n, 0..*, 1..*, m..* (m and n are integers). Visually, a set is shown by an arc that connects all the edges that are part of the set. The multiplicity is drawn in the centre of the arc.

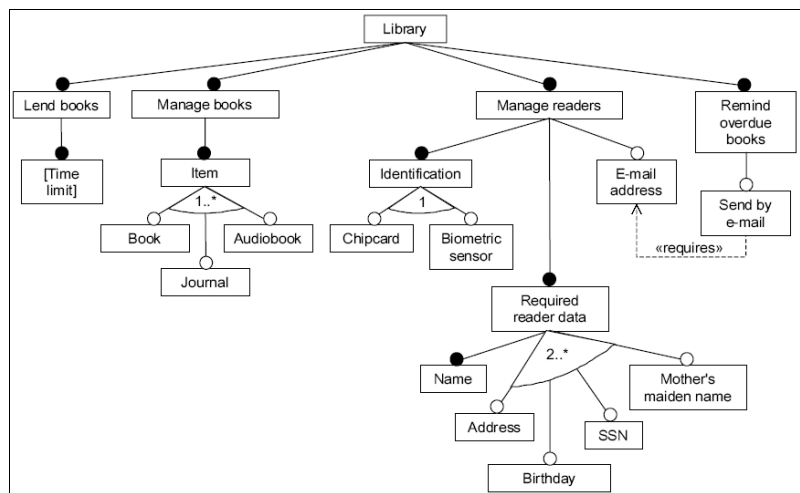


Figure 2.9 Riebisch et al. feature model for a library Product line, [Riebisch et al, 2002]

Furthermore, they propose that relations between features that are located in different not adjacent parts of the graph should not be shown on the feature model diagram because this reduces the clarity of the

diagrams. Instead, such relations can be described in a textual form rather than in the feature model. Figure 2.9 shows a feature model for a library product line represented in the Riebisch et al. notation. Note that the filled circles denote mandatory features, while the hollow circles denote optional features. Hollow circles for features in a feature group only indicate the direction of the decomposition. No distinction is made between OR and Alternative features, the feature model holds multiplicities for feature groups.

2.4.5 PLUSS

FODA originally introduced mandatory, optional, and alternative relations between features. PLUSS [Eriksson et al., 2005], which is the Product Line Use case modelling for Systems and Software engineering, introduced the notation of *multiple adaptor* to overcome the limitation of not being able to specify the *at-least-one-out-of many* relation in FODA. PLUSS also renamed alternative features to *single adaptor* features following the same naming scheme. The modelling notation was also slightly changed in PLUSS to meet the needs of the modified model, yet it remained a hierarchical tree structure based on the notation of FODA.

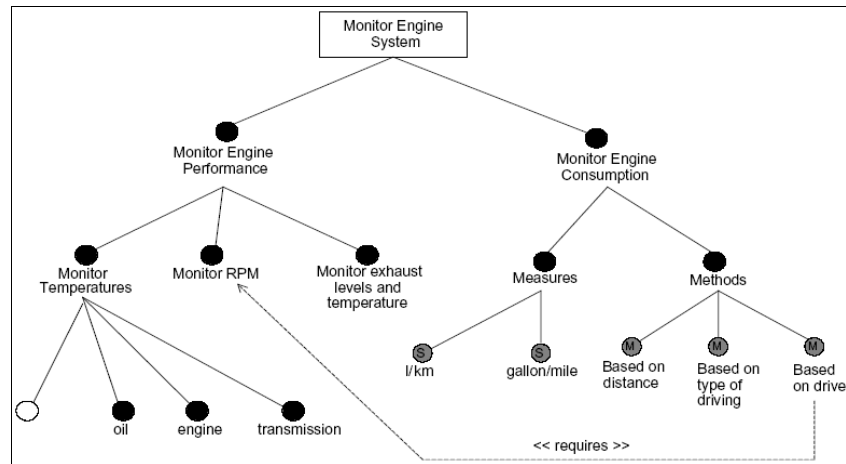


Figure 2.10 PLUSS feature model for a Motor Engine System product line, [Schobbens et al., 2007]

PLUSS represents mandatory and optional features similar to FODA; a filled black circle represents a mandatory feature and a non-filled circle represents an optional feature. It introduces new visual constructs to represent single adaptor and multiple adaptor features. Single adaptor features are represented by the letter 'S' surrounded by a circle. Multiple adaptor features are represented by the letter 'M' surrounded by a circle. Similar to FODA, the *excludes* and *requires* dependencies originally defined in FODA are used to represent constraints between features. Figure 2.10 shows a PLUSS feature model for a Motor Engine System product line.

2.4.6 Cardinality Based Feature Models

Cardinality Based Feature Models (CBFS) [Czarnecki and Kim, 2005] associate the concept of cardinality with each feature in the feature model. A feature model then represents a hierarchy of features, where each feature has a *feature cardinality*. Two types of cardinality are defined: *clone cardinality* and *group cardinality*. A feature clone cardinality denotes how many clones of the feature (with its entire subtree) can be included in a specified configuration. A group cardinality is an interval of the form $[m..n]$, where $0 \leq m \leq n$, and m and n are integers that denote how many features of the group are allowed to be selected in a certain configuration. Features still had one of four feature types AND, OR, Alternative, and Optional.

In addition, the notation of *feature attribute* was defined. A feature attribute indicates a property or parameter of that feature; it has a value that could be a numeric or string value. At most one attribute per feature is allowed. If several attributes are needed, a set of subfeatures, where each subfeature having an attribute, can be introduced. Additionally, a feature attribute value could be a reference to another feature; in this case it is called a *feature reference attribute*

The notation of FODA was extended to add the indication of cardinality and add a new notation that represents feature attributes. Two types of constraints are allowed, constraints between features and constraints on the value of the feature attributes (expressed in OCL¹⁴). The dependencies *implies* and *excludes* are used to represent constraints between features. Figure 2.11 shows a sample cardinality based feature model for an E shop product line.

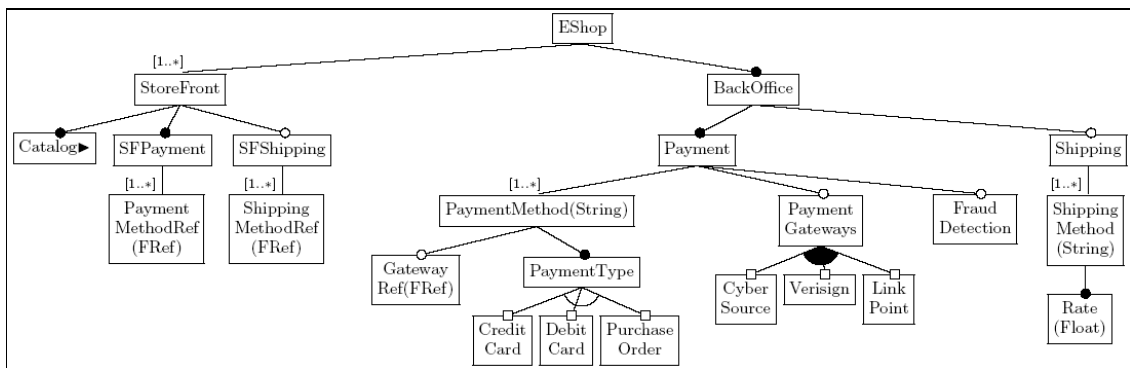


Figure 2.11 CBFM feature model for an E shop product line, [Czarnecki et Kim, 2005]

2.5 Feature Modelling Methods based on UML

UML (Unified Modelling Language) is a well-accepted modelling language for modelling software applications. Therefore, there were several proposals for extending UML to support variability modelling. UML variability modelling techniques use the concept of *class* rather than the concept of *feature* to model domain concepts and product line requirements. They model variability via adding variability profiles to UML for representing variability with UML models (e.g., [Claus, 2001], [Ziadi et al., 2003], [Gomaa, 2005]). These techniques aimed linking the variability of the domain (and later application) with the different UML models created at design time. We list here some of the most well-known attempts to extend UML to support the representation of variability information.

2.5.1 Claus UML Variability Stereotypes

Claus [2001], introduced two stereotypes to model variability, namely: `<<variationpoint>>` and `<<variant>>`. These stereotypes can be applied on any UML element that holds variability, i.e. a class, a component, a property or a package. It also applies to UML elements that hold behaviour such as collaborations, associations and methods. In order to differentiate variation points from each other, each variation point is given a unique name. This name can also be used to refer to that particular variation point in the documentation. A variation point implies some tagged values determining the binding time and

¹⁴ Object Constraint Language, <http://www.omg.org/spec/OCL/>

multiplicity of variants. The latter determines how many variants can be bound at binding time. The usage of each variant can be formally specified in a condition that determines when the variant is to be included at configuration time. Similarly variants contain a tag that relates them back to their variation points. For each variant a tag that holds the name of its parent variation point is added. In the meantime, feature interactions are modelled with dependencies similar to the ones defined in feature models. Two stereotypes are used to model dependencies <<requires>> and <<excludes>>. Additionally, the stereotyped <<evolution>> is used to represent evolutionary constraints between elements.

2.5.2 Ziadi et al. UML Variability Profile

Ziadi et al. [2003] defined a UML Profile which contains stereotypes, tagged values and constraints and which extends the UML meta-model to represent and model variability. These stereotypes are applied only to UML class diagrams and sequence diagrams. The stereotype <<optional>> is used to indicate a class that is optional, i.e. can be omitted in some products (similar to optional features in feature models). The representation of generalization/specification relations which denote variability is done using UML inheritance and stereotypes. Each variation point will be defined by an *abstract class* and a set of subclasses. The abstract class will be defined with the stereotype <<variation>> and each subclass will be stereotyped <<variant>>. An OCL constraint is defined such that each variant belongs to only one variation point.

The authors also make a distinction between the variability stereotypes defined for class diagrams and those defined for sequence diagrams. For variability in sequence diagrams, two stereotypes were introduced namely <<optionalLifeline>> and <<optionalInteraction>>. Optional objects within the sequence diagram are specified using the stereotype <<optionalLifeline>>, while the stereotype <<optionalInteraction>> identifies optional interactions between objects. Additionally, the stereotype <<variation>> indicates that the interaction is a variation point with two or more interaction variants. The stereotype <<variant>> indicates that the interaction is a variant behaviour in the context of a variation interaction. Constraints between classes are modelled using OCL.

2.5.3 Gomaa Variability Metaclasses

In [Gomaa, 2005] another attempt was made to combine UML and feature models. Features are modelled as metaclasses, UML stereotypes are used to represent the different types of (variable) features that are supported by FODA. Additionally some additional feature types were also introduced to the model to increase its expressiveness. The stereotypes defined for feature types are: <<optional feature>>, <<parameterized feature>>, <<common feature>>, <<default feature>>, and <<alternative feature>>. <<optional feature>> defines an optional feature. <<parameterized feature>> defines a feature that takes a parameter as its value (e.g. a colour feature would take the actual colour selected at binding time). <<common feature>> defines a mandatory feature.

Feature groups map the OR and Alternative nodes in FODA, they are defined using the stereotype <<feature group>> with the following stereotypes to indicate the selection guidelines <<zero-or-one-of feature group>>, <<zero-or-more-of feature group>>, <<exactly-one-of feature group>>, and <<at-least-one-of feature group>>. A feature group is represented as a UML generalization/specification relationship. <<alternative feature>> defines an alternative feature within a feature group. <<default feature>> defines a feature that should always be selected from within a feature group. A note about this technique is that it

considers features as classes in the UML diagram (i.e. it assumes a one to one mapping between features and classes).

2.5.4 Korherr and List UML Variability Profiles

Korherr and List [2007] define a new variability profile for UML 2. Two stereotypes are defined to denote variability, `<<variationpoint>>` which defines a variation point and `<<variant>>` which defines a variant. Variation points and variants are represented via a UML generalization/specification relationship. A generalisation Set defines a specific set of generalisation relationships. The metaclass describes how a general classifier (or superclass) may be divided using specific subtypes. Furthermore it has two meta-attributes with Boolean values, namely *isCovering* and *isDisjoint*. If *isCovering* is true, then the generalisation set is *complete*, i.e. selecting a variant is obligatory, that is equal to multiplicity 1..*, otherwise it is *incomplete*, i.e. selecting a variant is optional, this is equal to multiplicity 0..*. On the other hand if *isDisjoint* is true, then the generalisation set is *disjoint* (i.e. the variants have an XOR relation), otherwise it is *overlapping* (i.e. the variants have an OR relation). The *excludes* and *requires* dependencies between variation points and variants are supported via the `<<excludes>>` and `<<requires>>` stereotypes respectively. Additional constraints such as parameter values or name value pairs can be added in OCL.

2.6 Summary

In this chapter, we gave a brief introduction on variable software and in particular software product lines and we gave an example of a possible software product line for the development of a family of related products rather than developing one product at a time.

We have discussed that although software product lines may ease the development process and leverage the quality of the developed software as well as reduce the cost of development and time to market, it adds another level of complexity to the software development process as one needs to deal with the different variants of the product line and their commonalities and variabilities. For this reason, we need to carefully plan and model variability at an early stage. For this, we focused on feature modelling, as this is the de facto standard for modelling variability in software product lines.

Feature models are used to represent commonality and variability in a certain domain, leading them to be well suited for the domain analysis and modelling of software product lines. We explained the principles of feature models for modelling variability and commonality in software systems. Additionally, we discussed the importance of the term *feature* in characterising capabilities of the software to be modelled, which adds more convenience to using feature models. We have also explained the main feature modelling techniques; most of them based on the original feature modelling technique FODA and how these techniques differ in their notation and semantics. Additionally, we showed some other techniques that use UML profiles and stereotypes for modelling variability in software product lines.

Chapter 3

Related Work

As already mentioned in the introduction of this thesis there are quite a number of topics that this thesis relates to. In this chapter we will review works related to the topics of this thesis. We will first start with the works concerned with *representing and analysing feature models*; which feature models are represented with knowledge representation techniques and used to automatically or semi automatically analyse feature models (section 3.1). Next (section 3.2) we will consider works on using *feature models for configuration* of software product lines, in which automated and semi-automated feature analysis techniques are used for obtaining the possible products derived from a feature model. We also consider works on *modelling with separation of concerns* (section 3.3) which is a principle that could be applied in many different ways; we deal with some works that investigate its application to software modelling. We also consider works that investigate how this principle has been applied in the domain of feature modelling. Next, we discuss works related to *model integration and consistency checking*, which are techniques to merge conceptual models and check the consistency of the merged conceptual model (section 3.4). Next, we mention the works that introduce the concept of *multiple product lines*, and on modelling multiple product lines (section 3.5). We also discuss the efforts done on relating variability in the application to *variability in data and data schema* (section 3.6). Finally, we conclude this chapter by listing the efforts on *visualization of feature models* (section 3.7).

3.1 Representing and Analysing Feature Models

In this kind of work, the focus is on the semantics of feature models in order to better understand the information a feature model holds, such as understanding the variability opportunities that the feature model represents, and checking its feasibility (i.e. there will exist some feasible products out of this model, without actually finding these products). In these works, feature models are translated via knowledge representation techniques to formal knowledge models that can be automatically or semi automatically processed.

Having no agreement on common semantics for feature models has lead Bontemps et al. [2004] to study the formal semantics of FODA feature diagrams and compare these semantics with the formal semantics of successive feature modelling techniques extending FODA. The problem of the different notations for feature models was raised, the aim of the study was to study the expressive power of these different feature modelling techniques. The authors list the different notations that exist in the feature modelling domain and point out that they add no expressiveness to the semantics introduced by FODA. The paper also identified that the lack of common semantics makes transforming feature models represented by one technique into another difficult (i.e. manual rather than automated). The authors extend their study [Schobbens et al., 2007] to define in formal semantics, the FODA notation and give a comparison of the semantics of FODA with other feature modelling techniques. They define the so-called *general semantics* of feature models, which is the common semantics in all feature

modelling languages. They denote additional semantics defined by different languages as syntactic sugar.

Wang et al. [2005] adopt a semantic web approach to represent feature models. An OWL (Web Ontology Language [Patel-Schneider and Horrocks, 2004]) based approach was applied to represent feature models having the semantics of FODA. OWL DL was used to represent features, their relations, and dependencies. Individual features were represented as OWL classes. OWL constraints were used to model feature relations and feature dependencies defined by the feature model. With this setting, the feature model semantics represented in OWL was inspected for its consistency¹⁵, therefore the approach allows both representing and verifying feature models. Given a certain feature composition, the approach can detect whether it is valid or not; if it is valid that means that its ontology representation is consistent, if it is not valid that means that its ontology representation is inconsistent. Furthermore, it can also present the OWL DL axioms that cause the invalidity of the model; these axioms represent the modelling concepts of the underlying feature model. Because features are modelled as first class citizens, these axioms are the axioms that lead the ontology into an inconsistent state. The authors do not provide a representation of feature models in general but rather they apply a transformation to ontology for each individual feature model representing a certain case. The authors used the Racer reasoner [Haarslev and Möller, 2003] to check the consistency of the ontology and thus of the feature model which it represents.

Fan and Zhang [2006] propose a Description Logic (DL) representation of feature models. The authors propose a translation of feature model semantics to DL axioms that map the semantics provided by the feature model. A knowledge base that denotes the corresponding feature model is created. Therefore, the consistency reasoning on the feature model turns into the consistency reasoning on the corresponding DL knowledge base. Every node (i.e. feature) in the feature model is translated to a DL concept *C*, every edge (i.e. relation) in the feature model is translated to a DL role *R*. The feature model edge decorations (i.e. node types and group relations) are mapped to DL terminological axioms (i.e. DL *OR*, *AND*, *NOT*). DL cardinality constraints are used to map feature model cardinality constraints. The corresponding DL model is then checked for consistency via the Racer reasoner [Haarslev and Möller, 2003].

Peng et al. [2006] provide an OWL ontology for feature modelling. The ontology provides aid in application oriented tailoring. The ontology classifies features based on several categories depending on the underlying business model (e.g., action, facet, and term). The basis is *action*, which represents the business operation. In order to provide more business details for actions, the concept of *facet* was introduced. *Facet* is defined as dimension of precise description for Action. An action can have multiple facets and the facets can be inherited along with generalization relations between actions. Dependencies between features are identified based on their action requirements. The following dependencies were introduced: *Use*, *Decide* and *ConfigDepend*. *Use* denotes the dependency on other features for its correct functioning or implementation. *Decide* indicates that execution result of an action can determine which variant of a variable action will be bound for its parent action. *ConfigDepend* represents configuration constraints, which are static dependencies on binding states of variable features. Decisions about feature binding times and constraints regarding binding are made part of the ontology.

Abo Zaid et al. [2009] presented a framework for representing, integrating and validating feature models by using OWL and SWRL. The presented framework consists of an ontology that formally provides a specification for feature models. The mapping from feature models to ontology was defined by considering the meta model of feature models as first class

¹⁵ The authors define “consistency” in the context of their work as the OWL DL ontology consistency, i.e. all the axioms in the ontology meet the constraints that make the ontology consistent.

citizens of the model. The Ontology defined the concepts that make up the feature model and the actual feature model representing a certain problem adhered to these concepts. The feature model meta model captured by the ontology was defined based on a large category of the semantics in existing feature modelling techniques. The authors defined a set of feature-based integration semantics to enable the integration of distributed feature models. In addition, the authors provide means to integrate segmented feature models and provide a rule based model consistency check and conflict detection. SWRL rules were used to implement the rules checking the consistency of the feature model. A Description Logic reasoner was used to evaluate the rules and infer extra interesting information regarding the variability of the software. Furthermore, the ontology contained rules that can extract variation points and variants in order to allow users to quickly find relevant variability opportunities.

All these works emphasize the need for the formal representation and analysis of feature models. In this thesis we have also realized this need and therefore provided a formal representation for the Feature Assembly Modelling technique presented in this thesis (chapter 6). We have also combined this with defining the FAM Ontology which provides knowledge representation semantics based on OWL for representing the feature Assembly Models (chapter 10) and capturing errors which they may hold.

3.2 Feature Models for Configuration

In this kind of work the emphasis is on finding possible configurations i.e. feasible products that could be derived from the feature model. Feasible products actually represent feasible solutions for the constraints represented by the feature model. From that perspective a feature model is erroneous if no solution is found or if it contains one or more dead features. Dead features represent features that are not present in any of the feasible solutions of the feature model. These techniques do not take into account the fact that a contradiction in the model (due to bad design specifications) may be blocking feasible or expected feature combinations. Unlike the works presented in the previous section in which the consistency of the feature model was investigated, in these works, a feature model is considered¹⁶ consistent if it has feasible solutions, and contains no dead features (i.e. features that are never encountered in a valid solution). Some techniques also take into account detection of false optional features (i.e. features that are defined as optional but occur in every valid solution).

Batory [2005] used iterative tree grammar and propositional formulas to represent feature models. A logic-based Truth Maintenance System (LTMS) [Forbus and De Kleer, 1993] and Boolean Satisfiability Problem Solver (SAT solver [Eén and Sörensson, 2003]) are used to propagate constraints and find all feasible solutions. The feature model is transformed into a set of propositional formulas which are fed to the LTMS solver. The LTMS solver finds the set of solutions that satisfy the given constraints. The LTMS solver provides possible configuration, and it determines also if a certain configuration defined by the user is feasible or not.

¹⁶ We consider this a necessary but insufficient condition for the consistency of a feature model. As shown with the car example presented in section 2.2 the logical correctness of the feature model is also an important and often neglected issue. This correctness is achieved when the different stakeholders involved in the modelling of the domain have a thorough understanding of the features and how they influence each other. Furthermore, this understanding should be explicitly and rigorously modelled and made available to all stakeholders involved at later phases of the SPL domain engineering and application engineering phases.

In order to find the set of feasible configurations, Benavides et al. [2005] transformed a feature model into a Constraint Satisfaction Problem, in which features represent the variables and the feature dependencies represent the constraints. Features are associated with values of {true, false}, which means that the feature exists in the final product configuration (i.e. true) or it will be omitted (i.e. false). A constraint solver is used to determine the feasible configurations of the feature model, which is the solution that satisfies all the constraints. Typically many solutions should exist. A fitness function is used to bias the solver to select solutions that contain some desired features. The selection of the fitness function is dependent on the application.

Janota and Kiniry [2007] use Higher Order Logic (HOL) [Shapiro, 2001] to formulate feature models. The authors defined a generic feature model meta-model that integrates properties found in several feature modelling approaches from the literature. Again, a mapping is defined to transform the information contained in a feature diagram into HOL formulas. A feature is considered as a record with a set of attributes, where each attribute models a property of that particular feature. Utilizing this definition, they define a feature configuration as the set of features that are selected and the values of their attributes. Subsequently, a feature model is defined as a function that determines the set of valid configurations. The resulted HOL formulas are feed into a Prototype Verification System (PVS), a HOL solver, which is used to find feasible configurations.

Asikainen et al. [2007] define a domain ontology for modelling variability in software product families (as mentioned in section 2.4.3.2). The ontology was implemented using the Kumbang language, which is a combination of UML OCL constraints and natural language. To find feasible configurations the knowledge in the ontology is translated to Weight Constraint Rule language (WCRL), and Smodels¹⁷ [Niemelä and Simons, 1997] is used to find the possible configurations.

In the last years many authors, followed this path of using feature models for deriving configurations, a literature review of those techniques can be found in [Benavides et al., 2010]. As already mentioned, in this thesis we focus on the creation and representation of feature models as part of the domain analysis, we do not consider configuration of feature models. We have added an overview of work in the context of using feature models for configuration of SPL for the sake of completeness.

3.3 Modelling with Separation of Concerns

The “divide and conquer”¹⁸ approach is known to reduce the complexity of a big problem to smaller problems that can easily be solved separately. The same concept is also applicable when modelling large systems. It is a well-known fact that focusing on one aspect at a time allows for a better in depth understanding of that aspect. To help understand and reduce the complexity of software modelling and development the term *separation of concerns* was defined to allow tackling one aspect at a time. Separation of concerns is defined as “the ability to identify, encapsulate and manipulate only those parts of software that are relevant to a particular concept, goal, or purpose” [Ossher & Tarr, 2001].

¹⁷ Stable models, <http://www.tcs.hut.fi/Software/smodels/>

¹⁸ A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem [Wikipedia, http://en.wikipedia.org/wiki/Divide_and_conquer_algorithm]

To maintain separation of concerns in the software development process the term *perspective* or *viewpoint* was introduced by Finkelstein et al. [1992] to identify a mental view or outlook of a portion of the system in software architecture modelling. Finkelstein et al. [1992] defined a *viewpoint* as a locally managed object or agent which encapsulates partial knowledge about the system and its domain and which therefore contains partial knowledge of the design process. In subsequent works, the idea of modelling the software architecture using perspectives was further investigated to show how adopting perspectives helps in efficient modelling of the software system. The works by Graham [1996], Woods [2004] and Nuseibeh et al. [2003] are samples of such work where abstraction via viewpoints was introduced for software architecture modelling.

Due to the possible large size of software product lines in addition to the complexity associated with adding variability, recently some attempts to apply the separation of concerns principle for variability modelling have been carried out to improve the stakeholder understanding of large and complex systems. Mannion et al. [2009] proposed a multi perspective approach for modelling variability, in which perspectives were defined based on stakeholders' concerns. Each stakeholder has his/her own point of view and different usage for the products in mind; this reflects how he/she defines the expected/required variability. Therefore, using this approach, stakeholders are able to maintain their own partial models about the domain and its variability. There are no guidelines to how the viewpoints are defined, it is totally problem dependent and is driven by the involved stakeholders. The authors define a viewpoint as a subset of a master feature model containing only the features (and their interrelations) that are relevant for a given point of view. Examples of such viewpoints are market-driven viewpoints, and technology-based viewpoints. Furthermore, the authors propose a set of rules for conflict detection and conflict resolving between the different features belonging to more than one viewpoint with different constraints and relations constraining their selection in the final configuration. The aim is to support the configuration process when stakeholders have conflicting goals. Our work also uses the term "perspective" but its purpose is different and we also provide guidelines on how to use the perspectives.

Tun et al. [2009] use the separation of concerns principle to relate requirements to feature configurations, for this purpose three different types of feature models were created Requirement feature model, Problem World feature model, and Specification feature model. Requirement FM (RFM) describes different requirements that can be satisfied by the product line, this model is "high-level". The Problem World feature model (WFM) describes the features in the system context by showing different physical settings in which the software system might be deployed. The Specification feature model (SFM) describes the feature of the software, reflecting software engineers' view of the system.

A different approach for separation of concerns was adopted by Dhungana et al. [2010], they propose that the modelling space should be structured, so that large product lines can be managed more easily. The approach depends on defining model fragments that model the subsystems defined during the model structuring phase. A model fragment is a partial model with defined dependencies to other model fragments. The fragments need to be merged to have a global overview of the complete model, while doing so the consistency of the overall model is also checked. It is up to the modeller to define the appropriate fragments.

Rosenmüller et al. [2011] apply the Multi-Dimensional Separation of Concerns principle [Tarr et al., 1999] to variability modelling, they define the multiple dimensions variability modelling technique which aims to provide a way to model different variability dimensions separately and to integrate variability modelling with software product line configuration. The authors define a variability dimension as a kind of variability that is important for a stakeholder. They name the following examples of variability dimensions: the execution environment of a program (e.g., the operating system, the hardware), the context at

runtime (e.g., time, space, the user, etc.), non-functional properties (e.g., security, quality of service), and implementation variability. Because their objective is to use feature models for the sake of configuration the authors propose a textual variability modelling language for variability modelling in which the variability dimensions are described in separate variability models.

Hubaux et al. [2011] use the separation of concerns principle to provide different stakeholder views or perspective on large feature models for the sake of facilitating the configuration process. They define a view as a streamlined representation of a feature model that has been tailored for a specific stakeholder, task, or, to generalize, a combination of such elements. Therefore they are defined based on the different stakeholder interests and goals. The objective of the defined views is to facilitate configuration by only focusing on those parts of the feature model that are relevant for a given concern.

Schroeter et al. [2012] use user specific concerns to create different perspectives to enable tailored stakeholders views on large feature models. The objective of their work is to simplify the configuration process by providing stakeholders with views on the feature models that relate to their concerns. They distinguish between a perspective and a viewpoint; they define a perspective on a domain feature model as a virtual view resulting from the aggregation of multiple views, where each view is dedicated to a stakeholder's concern. While a viewpoint is defined as a collection of related group views being permitted to form a valid perspective accessible to stakeholders. A view model is created to hierarchically relate the different viewpoints to each other. The modelling and derivation of perspectives is a conservative extension to feature models. Therefore, the view model is separate from the original domain feature model and a multi-perspective model is used to integrate both. Features are assigned to groups of the view model and viewpoints are identified to create perspectives in the application engineering process. In the configuration phase stakeholders choose a viewpoint by selecting groups from the view model reflecting their concerns.

In the previously mentioned feature modelling with separation of concerns techniques, the emphasis is on handling the complexity of deriving products based on large feature models by separating them into smaller feature models using the separation of concerns principle. There are no concrete guidelines on how features are defined in the different perspectives, viewpoints or views; it is totally up to the involved stakeholders as the objective is to aid these different stakeholders involved in understanding large and complex feature models for the sake of making correct configurations. In the Feature Assembly Modelling approach, we adopt separation of concerns to support stakeholders during the modelling process in order to reduce and understand the complexity. We have adopted the principle of separation of concerns by using the concept of viewpoints or perspectives as a guide for identifying the features in addition to their variability. We propose a set of possible perspectives, and provide guidelines on how to identify features belonging to this perspective. We also provide guidelines on how a new perspective may be introduced.

3.4 Model Integration and Consistency Checking

This kind of work stems from the need to partition large conceptual models. As feature models are a particular kind of conceptual model we first start with works on model integration in conceptual models. Then we provide some specific examples of how these techniques were applied to feature models. Once the models have been partitioned into different chunks for ease of modelling, there is a need to combine the chunked models in order to have a complete picture of the overall model. Furthermore, the integration process should also hold a check for the consistency of the integrated model. We start by listing some generic work on model

integration and next we give current work that applies similar techniques for integration of feature models.

Kolovos et al. [2006] define a model merging language named *Epsilon Merging Language* (EML), which is a rule-based language for merging models of diverse meta-models and technologies. First a check for matching is performed with a set of match-rules. Each match-rule can compare pairs of instances of two specific metaclasses and decide if they match and conform to each other or not. EML allows defining a mapping between the elements of the two models to be merged via two different types of rules: *merge-rules* and *transform-rules*. Merge rules define the elements to be merged from the source model to the target model, while transformation rules defines the elements that can be transformed from the source model to meet the meta model of the target model. The authors did not provide means to check the consistency of the merged model.

Sabetzadeh et al. [2007] introduce the problem of consistency checking in typed graph based models. They define a set of RML (Relational Manipulation Language) rules which validate the consistency of the resulted merged model. Furthermore, the concept of global model merging was introduced to indicate the merging of all the existing heterogeneous models and then performing the consistency check on the global merged model. The merging is done via a merging operator, which was defined to bring together individual models by equating their corresponding elements. By keeping proper traceability information, consistency diagnostics obtained over the merge are projected back to the original models and their relationships.

Segura et al. [2007] propose a model merging approach for merging segmented feature models using graph transformations. The authors propose that the segmentation of feature model could be in terms of both time and space. They define the process of merging feature models as an operation that takes as input a set of feature models and returns a new feature model representing, as a minimum, the same set of products than the input feature model. The proposed technique allows for automating the merging of feature models based on a set of merging rules. The authors define a catalogue of visual rules to describe the possible different merging conditions and show with each condition the merging result.

Acher et al. [2009] define operators for merging feature models, two composition operators *insert* and *merge* are defined. *Insert* is used to insert features from one feature model into the other. This is done by introducing newly created elements into any base element or inserting elements from the aspect model into the base model. The proposed insert operator supports different ways of inserting features from a crosscutting feature model into a base feature model. *Merge* is used to combine matching features in two feature models in order to obtain a feature model that combines the two feature models. As a result of the merge some features may be renamed or deleted to achieve the consistency of the resulted feature model. Acher et al. [2012] extend their work by proposing creating fragments of feature models to overcome the large size and complexity of the one feature model paradigm, they propose using their merging operators to merge these many feature models.

In the work presented in this thesis, we use a technique similar to the merging operator(s) defined by Acher et al. [2009] to link related feature together based on feature dependencies.

3.5 Multiple Product Lines

Holl et al. [2012] define a Multiple Product Line (MPL) as a set of several self-contained but still interdependent product lines that together represent a large-scale or ultra-

large-scale system. The different product lines in an MPL can exist independently but typically use shared resources to meet the overall system requirements. Multiple product lines are referred to in the software product line literature using many terms of which are ultra-large-scale systems [Northrop et al., 2006], software ecosystems [Bosch, 2009], compositional product lines [Bosch, 2010], or product populations [van Ommering, 2000]. Several scenarios exist where there may be a need for multiple product lines. The first scenario is that starting with one product line diversity increases among the features that it becomes more flexible to consider several product lines of less diversity and size. According to Van Ommering [2002] in real case situations the scope and diversity of the product line increases leading it to be split to several product lines instead of one in order to better manage its increasing size and diversity. The speed on which a product line evolves to a multiple product line depends on customer requirements, the market needs, technology requirements (and also offers) and innovation support. The second scenario is that the product line is too large because it models a complex and large system in which each part on its own may also be considered a product line. Describing this scenario Northrop et al. [2006] indicate that in very large systems both variability and complexity can be better managed by splitting these systems to multiple product lines. The third scenario is actually a combination of the first two scenarios and motivated by the need for openness in software development (e.g. for outsourcing and using third party components). Describing this scenario, Bosch [2009] indicates that as the diversity of the product line increase and there is a need for openness to third party development then software ecosystems is the next logical step for a company that has a successful platform and intra-organizational software product line.

Many works have investigated the support for modelling such product lines for example the work by van Ommering [2002] investigates the use of component based architecture models for maintaining reuse of components (which are also features) between the different product lines. Bühne et al. [2005] propose a meta model for structuring variability information in the requirements artefacts across product lines. The presented meta model is based on the OVM-notation (Orthogonal Variability Modelling [Pohl et al., 2005]). Hartmann and Trew [2008] support the modelling of Software Supply Chains which represent a Multiple Product Line. The multiple product line is modelled combining a Context Variability Model (contains classifiers of context) with the conventional feature model to create a new model, which they call the Multiple Product Line Feature Model. Features are related to context classifiers through feature dependencies. According to Reiser and Weber [2006] multiple product lines is a typical case in the automotive industry where several product lines exist overtime. The new product lines are initially taken from existing product line models amending them to new requirements. The authors introduce the notion of *reference feature models* and allow traditional feature models to be enhanced with the option of having such a reference feature model. Thus are called the referring feature model. The reference model serves as a template and guideline for the referring model by defining default features together with their default properties and by defining which deviations from these defaults are allowed. A hierarchical organization of product sub-lines is composed to reflect how the products relate to each other. Based on this hierarchy a multi-level feature tree consisting of a tree of feature model in which the parent model serves as a reference feature model for its children. In recent years of research on the topic more attention was given to the architecture support and configuration of multiple product lines, interested readers may refer to the recent survey of Holl et al. [2012] on capabilities supporting multi product lines.

In this thesis, we also take into consideration that multiple product lines can spring from the first initial product line, in which some features may have different variability specifications (e.g. based on market, legal or technology requirements). However, instead of expanding the concept of software product line to capture multiple product lines, we allow to create different product lines (as well as different products) from existing features by

supporting reuse of features as early as the domain analysis modelling phase. We tackle the issue of reuse of features at the feature modelling level in chapter 9.

3.6 Variability Modelling and Databases

Development of efficient data intensive software product lines requires an alignment between the features of an individual product and the data (governed by a data schema) on which these features operate. Different features may require different parts of the data. Tailoring the database to the specific needs of a database actor is a well-known issue in database design. Often many views are created to suit the specific needs of different users or user groups (i.e. actors). For example, Nyström et al. [2004] proposed *schema tailoring* to meet the needs of different actors accessing different portions of data, in different usage scenarios. Data views were tailored for different actors in different contexts. It amounts to cutting out the appropriate data portion that fits each possible actor-context.

The issue of matching the database with each member of the product line was first raised in embedded systems [Tesanovic et al., 2004], [Rosenmüller et al., 2008] and [Rosenmüller et al., 2009], where the hardware is diverse and only limited resources exist. Therefore it is very important that the application and its accompanying database, as well as the database management system are suitably tailored to meet the different requirements. Tesanovic et al. [2004] perform the tailoring process at runtime to provide a configurable real-time database platform (for both data and DBMS). Rosenmüller et al. [2008] adopt a product line approach to develop both the application and the suitable database management system (and also data) for each product. In that case, it was crucial that with each product of the product line only the essential data management requirements and essential data existed. The focus was given to the variability of the DBMS features. The authors did not mention how the database entities were affected by this variability in DBMS features. Rosenmüller et al. [2009] extend their work and use a feature oriented programming approach for tailoring a DBMS for embedded systems, in which a feature model describing the DBMS features and their variability. Feature oriented programming was used to create a common architecture and code base that allowed to configure different configurations of the DBMS (the approach was applied to Berkeley DB).

Bartholdt et al. [2009] propose an approach for *Data Model Variability (ADMV)* in which a feature model is created to model the variability and commonality of the software product line, a data model (represented in UML) is also independently created to represent the data entities of the software product line. The ADMV addresses three types of variability: positive - adding new fields, data or relations to the core model; negative - eliminating fields, data, or relations from the core model; and structural - varying the type, cardinality, or naming of elements. The stereotype <<*Variation*>> is used to define the variable types in the data model. The modelling of variability and data in a central model makes the effects of the variability more traceable. This approach is close to our approach, yet they provide no guidelines for how the variability in the data model was defined (like we do) and whether its relation to system features were considered early in this process or not.

Siegmund et al. [2009] propose to tailor database schemas according to user requirements. Two methods were proposed, *physically decomposed schemas* (i.e. physical views) and *virtual decomposed schemas* (i.e. virtual views) for representing variability in the application and matching this variability with variability in the corresponding database. Once a product is configured (i.e. the features of the product are identified) the schema is tailored to meet the needs of the product features. The proposed technique decomposes an existing database schema in terms of features. It allows tracing of the schema elements to the program

features at the code level using a technique similar to the `#ifdef` statements of the C preprocessor. The presented approach focuses on the tailoring process of the schema, the description of how the variability in application features is related to data entities is not detailed.

In this thesis we present an approach to support modelling the variability and commonality of the data according to the variability and commonality of the application. Variability of the application is analysed in order to derive a variable data model maintaining traceability links between the variable features of the application and their corresponding variable entities in the database. This approach is presented in chapter 7.

3.7 Feature Model Visualization

Adopting visualisation techniques in software product line engineering can help stakeholders in supporting essential work tasks and in enhancing their understanding of large and complex product lines [Nestor et al., 2008]. The purpose of the visualization may vary from providing cognitive support for understanding the complexity in the product line to aiding the configuration process.

The V-Visualize tool [Sellier and Mannion, 2007] uses force directed layouts to represent variability represented in decision models and inter-dependency models. Decisions are represented as nodes and dependencies are represented as edges in the proposed visualization. Nestor et al et al. [2008] propose a set of guidelines for providing visualization support for managing variability. Based on these guidelines they propose the Visual and Interactive Tool for Feature Configuration utilizing a simple non-radial tree layout with support for colour encoding of information and details on demand support [Cawley et al., 2008]. FeatureMapper [Heidenreich et al., 2008] provides tree-based visualization support for developers in understanding the mapping between features from a feature model and their realisation in solution models. The tool provides four views for this, the Realisation View, the Variant View, the Context View, and the Property-Changes View. The objective of the tool is to support users in configuring large and complex product lines. Trinidad et al. [2008] propose Feature Cone Trees (FCT) visualization of feature diagrams using cone trees as an alternative to represent large hierarchies in the three dimensional space. Interaction with feature cone tree is done by rotating all the sub-cones whenever a node is selected. Cawley et al. [2009] propose a 3D visualization for variability models, the variability models considered are decision models, feature models, and component models. The proposed visualization consists of three graph axes; the decision model is mapped to the Y-axis, the feature model to the X-axis and the component model to the Z-axis. The mapping is a sequential listing of the model elements along an axis.

In this thesis, we use a force directed graph for visualizing the Feature Assembly models, users can browse the visualized models on demand. For the ease of browsing colour schemes, which can be adjusted by the users, are supported. As not all the information is important to all users, users are allowed to indicate which information they wish to see on the visualization. Query support is also provided.

3.8 Summary

In this chapter we have reviewed relevant work related to the topics of this thesis. We have started with the works concerned with representing and analysing feature models; feature models represented with knowledge representation techniques, and automatically or semi

automatically analysed feature models. Although out of the scope, but for the sake of completeness, we listed some works on using feature models for configuration management in which stakeholders are concerned with finding the possible products derivable from the feature model(s). Next, we mentioned some of the early works on modelling with separation of concerns in software engineering in general. We also considered the works that investigate its application to feature models. Next, we mentioned works related to model integration and consistency checking, which are techniques to merge conceptual models and check the consistency of the merged conceptual model. We also have considered the work done on multiple software product lines. We described the efforts done on linking the variability in the application to variability in data and data schema. And finally, we concluded this chapter by looking to the efforts on visualization of feature models.

Chapter 4

Background

In this chapter¹⁹ we give some background information related to the domain of knowledge management as we adopt a knowledge management approach for representing and managing the information about the commonality and variability of the features representing a specific software product line. Knowledge management and knowledge representation is a broad discipline; in this chapter we present only the parts that are relevant to the work presented in this thesis.

Knowledge is defined by philosophers as a meaningful resource that makes us knowledgeable about the world. Theories of knowledge define what is about the world, how is it encoded, and in what way we reason about the world [Lim et al., 2011]. In the context of computers and information systems the term *Knowledge* is used to define *meaningful information*. Knowledge is defined by Schreiber et al. [2000] in the context of information systems as “the whole body of data and information that people bring to bear to practical *use in action*, in order to carry out tasks and create new information. Knowledge adds two distinct aspects: first, a sense of *purpose*, and second a *generative capability*”. An important aspect about knowledge is that it cannot be looked at in isolation. Knowledge is only important within a certain context. What could be important knowledge in one situation could be useless in another. This is why knowledge is said to have a sense of *purpose*. Understanding the purpose of the knowledge means defining when it will be used and by whom. Different people may have different perception on the same piece of knowledge. Additionally, different people may require different views of knowledge as well as different levels of detail. Too little knowledge for one could be too much for another. Therefore, for an efficient understanding and utilization of knowledge, knowledge management techniques should be adopted. *Knowledge management* is the discipline under which information is turned into actionable knowledge and made available effortlessly in a usable form for people who can apply it [Kimiz, 2005]. Knowledge management is a process that consolidates three essential phases [Schreiber et al., 2000] [Prece et al., 2001]:

6. **Knowledge Acquisition:** This is the process of finding out what knowledge needs to be managed. The knowledge that should be represented and manipulated by the intended knowledge management system should be captured; its scope should be defined. Knowledge acquisition includes an understanding of the problem domain(s) to which the knowledge belongs. At this point, a thorough understanding of all the essential information within the problem domain should be achieved. Additionally, an understanding of which information is required by whom (this could be a person or another system), how it is sought (i.e. what type of queries that the user is likely to issue to the system), and at which level of details.
7. **Knowledge Storage:** Once captured, knowledge should be well defined and represented. A model is created to represent the knowledge captured in phase 1, often referred to as a

¹⁹ Readers familiar with knowledge management can skip the chapter.

conceptual model. A conceptual model is an abstraction of some part of reality; it is conceptual and independent from the actual form of implementation. Furthermore, a conceptual model provides a formal description of the knowledge within the problem domain. While doing so certain decisions should be made to provide the right scope of the conceptual model and what concepts within the domain are of interest. This process is based on the knowledge captured during the knowledge acquisition phase. Conceptual models provide a medium for explicitly and formally representing knowledge. Yet they remain at a conceptual level and represent a starting point for understanding, abstracting from any implementation. In order to obtain a knowledge model that provides solutions that answer the needs of the users, a processable knowledge model is required. To obtain a processable knowledge model the conceptual model is represented by means of a *knowledge representation technique* to obtain a physical model represented in a chosen knowledge representation structure. The resulted processable knowledge model is a formal unambiguous representation of the conceptual model that can be further stored and manipulated. Because this phase is the most important phase for the success of any knowledge management system, it will be discussed in details in sections 4.1 where we discuss conceptual modelling and in section 4.2 where we discuss knowledge representation techniques.

8. **Knowledge Manipulation:** Once a processable knowledge model is built, the knowledge management system is ready to use. Users can use the system to retrieve stored information or deduce new information from already existing information (through inference techniques, as will be discussed later). Additionally, users can query the system for parts of some specific knowledge (e.g., via queries).

4.1 Conceptual Modelling

Knowledge modelling is the act of building abstract knowledge models to represent already existing real world systems in a comprehensible and formal manner. At an early stage of knowledge modelling, knowledge models are built independent from any implementation issue. Therefore this stage is often referred to as *conceptual modelling*. The knowledge model is then known as the *conceptual model*. Conceptual modelling is one of the key topics in information systems (IS) [Wand and Weber, 2002]. Conceptual modelling is also considered a crucial activity in software engineering [Dieste et al., 2002]. Conceptual models are abstractions describing the world from a conceptual point of view, while doing so they hide certain details while illuminate others. Conceptual models assist in understanding the world to be modelled in three essential ways [Allemang and Hendler, 2008]:

- **Conceptual models help people communicate.** A conceptual model describes the situation in a particular way that other people can understand.
- **Conceptual models explain and make predictions.** A conceptual model relates primitive phenomena to one another and to more complex phenomena, providing explanations and predictions about the world.
- **Conceptual models mediate among multiple viewpoints.** No two people agree completely on what they want to know about a phenomenon; conceptual models represent their commonalities while allowing them to explore their differences.

Conceptual models represent information using semantic terms, such as entity, relationship, concept, event, goal, etc., and semantic relationships, such as roles, and associations [Mylopoulos, 2001]. Concepts may be organized into concept hierarchies by means of their generalization-specification relationships. Or they could be organized in terms of part-of relationships (i.e. aggregation). Furthermore, a conceptual model should also define

how concepts relate to each other in terms of their intersection and/or union relationships. Moreover, a concept may be disjoint with one or more concepts to indicate that it cannot intersect with it (i.e. to explicitly mention in the model that the two concepts are totally different).

A conceptual model should represent the knowledge within the domain it represents using the terminology used in that domain. Domain terminology is referred to as *domain vocabulary*. It is important to bear in mind that no knowledge model is capable of providing an exact and complete representation of a specific domain, nor should it. This is because knowledge is a relative thing, i.e. not all the knowledge in the domain is significant for a certain purpose, rather only the knowledge relevant for the problem being modelled from a specific context (the context that the knowledge model is created for). For example when modelling a car many concepts can be defined. Adopting a component wise context, broad concepts like: *Engine*, *Wheels*, *Transmission*, and *Chassis* can be defined, while adopting a visual characteristic context, leads to defining concepts such as: *Colour*, *Wheels Type*, *Model Number*, *User Age Category*, and *Number of Seats*. For a system that mechanically simulates a car the first set of characteristics are far more important than the second set. While for a car selling customization system, the second set is more important (the first set is fixed for a specific car). In the case of building a car buying recommender system both sets of concepts become important.

Therefore, it is important that any conceptual modelling process should start with defining the purpose for the modelling. This is done by answering the question “*what is the model intended for?*” Answering this question should help in anticipating what domain concepts should be relevant and why. Next, all the relevant domain concepts found in the real world knowledge model should be mapped to their representative concepts (and sometimes also attributes) in the corresponding knowledge model. Furthermore, the conceptual model should also capture rules or restrictions governing the definition of the concepts of the domain. This kind of knowledge is known as *tacit knowledge* [Schreiber et al., 2000]. It is often not found explicit in the domain but comes from understanding the domain and how the different activities are done within that domain. This intrinsic knowledge and hidden rules that govern how the different domain concepts interact should be made explicit and should become part of the conceptual model.

Figure 3.1 shows the process of conceptualizing knowledge from real world to a knowledge model representing the problem domain. Figure 3.1 also shows that the conceptual model serves as a first prototype for the actual processable knowledge model [Wielinga et al., 1992].

For the sake of formally representing and communicating knowledge, different conceptual modelling languages have been proposed, each providing a meta model for representing domain concepts. Conceptual modelling languages are mostly visual languages to increase the cognitive ability of domain experts and to facilitate communication

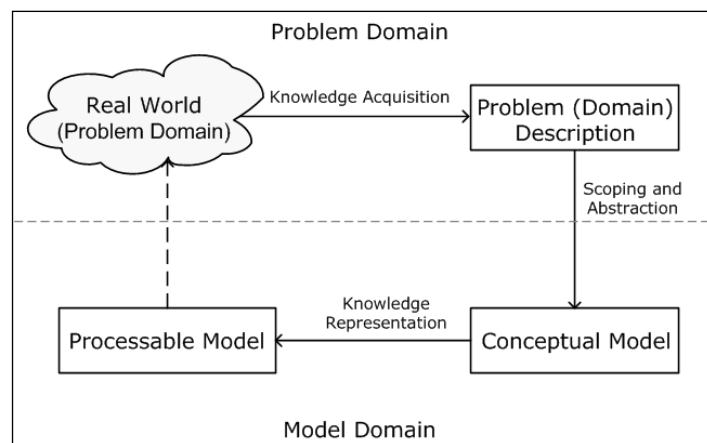


Figure 4.1: Conceptual Modelling Process [modified after Kotiadis & Robinson [2008]]

and cooperation. We list here some of the most common conceptual modelling languages and a brief description of their meta model concepts, namely: ORM [Halpin and Morgan, 2008], ER [Chen, 1976] and UML [OMG].

- **Object Role Modelling (ORM):** ORM is a fact oriented modelling technique; it conceptualizes the domain in terms of objects and their roles. Objects are the concepts of the domain, and roles identify the roles objects can play in relationships between these concepts. ORM also provides specialization relationships by means of subtypes and provides a rich set of constraints to express the rules that apply in the domain. Furthermore, ORM structures may be directly verbalized as sentences, it is based on few orthogonal constructs, and it reveals semantic connections across domains.
- **Entity Relationship Modelling (ER):** ER describes the domain in terms of entities and their relationships. Entities represent the domain concepts which could have associations that link them together, these associations are the relationships. Furthermore, entities are further associated with additional information which identify properties of these entities, these are referred to as attributes. Together entities and relationships provide a mapping of the domain concepts and how they are related together. Attributes provide more details about these domain concepts.
- **Unified Modelling Language (UML):** UML on the other hand is an object oriented modelling technique that was originally intended to model software, and also found its way to data modelling (class diagrams). It models the world in terms of objects and their relations namely associations and generalizations. UML's class diagrams also allow modelling *attributes* (properties) and *operations* (behaviours) of an object. Each object has a type which is defined by means of a *class*. A class defines the properties and behaviours of its objects and also its relationships.

Each of these languages (ORM, EER, and UML) is based on a different theory of representing knowledge, i.e. each has a different set of associated syntax (marks), semantics (meaning) and pragmatics (use) [Halpin and Bloesch, 1999].

4.2 Knowledge Representation Techniques

As mentioned above, a conceptual model provides a concrete understanding of (a part of) the real world, i.e. domain, abstracting from any implementation issue. Once a conceptual model is created, this model should be used to obtain a processable model, which is referred to as a *knowledge model*²⁰ also called a *computational model*²¹. By a processable model we mean one in which knowledge can be made available and accessible in a comprehensible manner for both humans and machines (hence the name computational model). For example, it can be queried to retrieve information, i.e. facts stored in the model. Furthermore, a knowledge model should not only represent and store facts about the domain, but also provide the necessary information to reason about these facts. Therefore, it is important to use unambiguous terms for representing the domain concepts and how they are related, in order to allow useful inferences to be made. *Inference* refers to the ability of deducing new knowledge from existing knowledge. How new information should be deduced is a fundamental part of any knowledge model. Without the model explicitly stating the situations that should trigger inferences and the rules that the inferences should follow, no inferences can be made. This is actually an important part of the model when modelling a system for problem solving.

²⁰ This is the typical naming used in Information Systems.

²¹ This is the typical naming used in Artificial intelligence.

Failing to identify correctly relationships between the concepts within the model will lead to a misinterpretation of the system. Likewise failing to identify correct rules that guide the inference process will also yield in wrong inferences. The resemblance between the facts in the model and these facts in the domain is referred to as the *fidelity* of the model [Davis et al., 1993]. A correct model is one that achieves the highest possible fidelity, in both representing the facts of the domain and the rules that govern how new facts can be deduced. This ensures that the inferred information is always true.

According to Schreiber et al., [2000] a knowledge model should contain three groups of knowledge, *domain knowledge*, *inference knowledge* and *task knowledge*. Domain knowledge is the knowledge about the specific domain. It represents the domain concepts, their attributes, and the relations (e.g., classification, aggregation, etc.) between these different domain concepts. Domain knowledge should capture both explicit and tacit knowledge in the domain. For example for specifying a car the concept *wheel* can be defined, which has an attribute *diameter* and a constraint that a *car* has four identical wheels. Inference knowledge on the other hand is the set of specifications by which new knowledge can be inferred from existing one. Furthermore, inference knowledge identifies the knowledge roles of interaction between the different (static) domain concepts. These roles act as functional transfers that show knowledge which is related to some activity. For example within a car buying customization system the inference *has colour* could be used to relate a certain car with a specific colour. Additionally, inferences are also used to represent rules within the system; as an example a *customized car* is a car that has a specific *colour*, specific *seat number*, a specific *wheels type* and a valid *model number*. Task knowledge is the knowledge concerned with the goals of the knowledge system, it tries to answer the question “for *what was this knowledge model developed?*”. Task knowledge identifies a hierarchal decomposition of tasks that act as a solution to realizing a certain task. In the work presented in this thesis we are only concerned with domain knowledge, inference knowledge (as will be discussed in chapter 10).

Developing a knowledge model for a certain problem is not a straightforward task; rather it is an iterative task that depends greatly on the purpose of the model. According to Bylander et al. [1988] representing knowledge for the purpose of solving some problem is strongly affected by the nature of the problem and the inference strategy to be applied to the problem. Therefore, a variety of techniques have been presented to represent knowledge models, referred to as *knowledge representation techniques*. Each one of these techniques has its own semantic capabilities (i.e. expressive power) and inference capabilities. Based on one or more of these techniques different languages have been defined to represent knowledge. Some of the most common techniques are: logic-based knowledge representation, rule-based knowledge, semantic networks, and ontology. We will discuss each in more details.

4.2.1 Logic Based Knowledge Representation

Logic based knowledge representation is probably the most common and widely known technique to represent information since the development of knowledge representation techniques in the early 1970's. The popularity of logic is due to its capability of unambiguously representing facts about the world. The most popular species of logic for knowledge representation is First Order Logic (FOL); this is due to its high expressive power. The basic elements of the representation are characterized as unary predicates, denoting sets of individuals, and binary predicates, denoting relationships between individuals [Baader et al., 2003]. For example, statement (4.1) below shows a first order logic statement stating that for every variable *y* that is a *Car*, it is also a *Vehicle*. Statement (4.2) states that *Car2* is a *Car*. Reasoning in first order logic is mainly used to check consistency of the defined premises.

Furthermore, first order logic supports querying the existing knowledge, by identifying whether a given premises is true or false. An important note about first order logic systems is that the defined models may not be finite depending on the complexity of the defined premises. In this case it is the modeller's task to validate whether or not the defined system is finite.

$$\forall y(Car(y)) \rightarrow Vehicle(y) \quad (4.1)$$

$$Car(Car2) \quad (4.2)$$

Description Logics (DL) (a second order logic) was defined in the 1980's. It gained popularity as a knowledge representation technique due to being concept-based rather than functional-based as in first order logic. In description logic, a distinction is made between domain concepts and individuals that belong to the domain [Baader et al., 2003]. Domain concepts are perceived as *terminology* within the description logic representation of the world; it is referred to as the *TBOX*. The *TBOX* holds the declarations that describe the concepts of the domain and defines how these concepts are structured in a concept hierarchy. It also describes concept properties, which declare the relations between concepts. Knowledge concerning the declaration of individuals in the domain is known as assertions and is referred to as the *ABOX*. DL statement (4.3) below states that *Car* is subclass of *Vehicle*, i.e. all *Cars* are *Vehicles*. DL statement (4.4) states that *hasColour* is a property of the *Car* concept; it identifies the colour of a certain car. Statement (4.5) is an assertion that *Car 1* has a *Red* colour. Statements (4.3) and (4.4) are part of the *TBOX* while statement (4.5) is part of the *ABOX*. Knowledge represented in terms of DL logic representation can be formally reasoned about; three forms of reasoning are available, satisfiability, subsumption, and consistency check. If an expression is satisfiable it means that it is consistent with the knowledge defined in the DL system. Subsumption means identifying the hierarchical relation between the concepts. When reasoning for subsumption, a concept hierarchy is defined for all concepts part of the knowledge model in order to relate them to one another. Consistency check validates that there are no contradicting facts defined in the knowledge model. Furthermore, a DL system can be queried for individuals that belong to a certain concept or satisfy a certain premises.

$$Car \subset Vehicle \quad (4.3)$$

$$hasColour.Car \quad (4.4)$$

$$hasColour(Car1, Red) \quad (4.5)$$

Unfortunately, logic representation of knowledge is difficult to understand for non-logicians. Despite their expressiveness in representing knowledge, their usability is a major drawback.

4.2.2 Semantic Networks

Semantic networks are flexible and easy to use structures for representing knowledge. They can easily be created and read by non-logicians. A semantic network is a directed graph notation for representing knowledge in patterns of interconnected nodes and arcs [Kendal, and Creen, 2007]. The nodes represent individuals (i.e. knowledge objects or instances) and the arcs represent how these individuals are related to each other. An arc holds a name that represents the relationship (also role) which it holds with the individual it is connected to. Sowa [1987] identified six different types of semantic networks, which differ in their expressivity and formality of their representation.

Figure 3.2 shows a sample semantic network that represents the following information: *Car is a Vehicle*, *Car 1 is a Car*, *Car 1 hasColour Red*, and *Red is a Colour*.

Although semantic networks are very powerful in expressing knowledge they have two major drawbacks. Firstly, their flexibility in representing knowledge in various ways and using various vocabularies makes it difficult to represent exceptional cases. Secondly, they scale badly. Despite their flexibility and ease in representing knowledge, when the represented knowledge is large the semantic network grows in size and becomes difficult to read and to analyse.

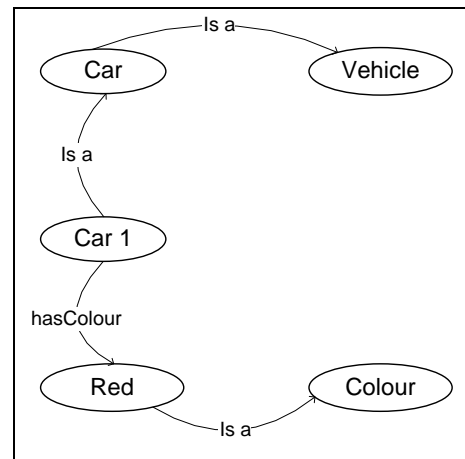


Figure 4.2: Sample Semantic Network

4.2.3 Ontologies

“Ontology” is a term coming from philosophy that means the study of being or existence. It refers to a system of categories to describe the existence of the real world, or the classification of being [Gruber, 2008]. The term ontology found its way to computer science due to its ability to describe the world using formal semantics. Ontologies were first used in Artificial Intelligence as a way of specifying content-specific agreements for the sharing and reuse of knowledge among software agents. Ontologies then found their way to other disciplines of computer science as a way to formally conceptualize knowledge within a certain domain allowing for a common understanding of that knowledge. By providing a formal and common understandable representation of knowledge in a certain domain, ontologies allow for knowledge sharing and knowledge reuse.

Gruber [1993] defines an ontology as *an explicit specification of a conceptualization*. A conceptualization is an abstract, simplified view of the world that we wish to represent for some purpose, i.e. a conceptualization is the universe of discourse or domain of interest. Therefore, a conceptualization refers to the formal representation of the concepts and relationships between these concepts with respect to a specific domain of interest. An agreement on a certain specification of a certain domain indicates how all agents committed to use this ontology should interpret the concepts of the domain. This allows these agents to have a consistent understanding of these shared concepts. This gives ontologies their power of promoting knowledge sharing.

Nowadays, ontologies are being used as some form of formal representation for the terms and concepts of a particular domain of interest in a particular situation or problem context. Therefore, ontologies are used to promote an agreement on some shared concepts. A more generic definition for an ontology is “*An ontology is an explicit specification of a shared conceptualization that holds in a particular context*” [Schreiber, 2008]. As already mentioned, the purpose of an ontology is to describe facts assumed to be always true, by means of defining vocabularies which users of a certain domain agree with, as means to conceptualize this domain. The concepts of the domain are organized by means of concept hierarchies (or taxonomies) using the vocabularies of the domain. Furthermore, within these concept hierarchies concepts are related to each other by means of properties. Properties govern how each concept should behave within the domain, i.e. they define the roles of the concepts within the domain. Additionally, an ontology could define restrictions on concepts or properties to govern the relations between the concepts of the domain. Furthermore, rules could be added to indicate how additional knowledge within the domain can be inferred. Many ontology

languages exist to represent ontological knowledge but recently the Web Ontology Language (OWL) gained great popularity in providing formal and portable ontologies (more details in section 3.3.3).

Several types of ontologies exist depending on the purpose of the ontology. For example, ontologies can be used to describe generic domains these are often called *Foundational ontologies* (also called *upper ontologies*), they provide generic terms and concepts that can be used within other more specific ontologies, therefore allowing for knowledge reuse. For example, the Time ontology [Time Ontology in OWL, 2006] defines the temporal content of Web pages and the temporal properties of Web services. The ontology provides a vocabulary for expressing facts about topological relations among instants and intervals, together with information about durations, and about date-time information. Other such ontologies are the Workflow ontology [Sebastian et al., 2008], the Space ontology [space ontology, 2011], and the Basic Formal Ontology (BFO) [Grenon, 2003]. Domain ontologies are more restrictive; they are used to provide a more specific representation of concepts and relations within a certain domain in a certain context. Domain ontologies are the most common types of ontologies for the sake of knowledge representation for problem solving. They not only capture the terms and vocabularies used in a certain domain but also capture the restrictions that govern the relationships between the concepts of this domain. Additionally, they define the rules that define possible inferences. Examples of such ontology are: the Gene Ontology [gene ontology, 2011], the Pizza ontology [pizza, 2011] and the Petri-net Ontology [Gašević and Devedžić, 2006].

Designing an ontology should not be an ad-hoc task, several ontology-engineering methodologies exist to support the design of a well-formed ontology, each defining its own terminology for the concepts defined within the ontology. Within these ontology engineering methods, conceptual structures of a domain are conceptualized in terms of classes, properties and restrictions. Classes represent the real world concepts while properties represent the valid behaviour of these concepts, and restrictions represent the set of rules governing the relations between the concepts of the domain. Most ontology engineering approaches²² include the following phases for defining an ontology:

1. **Define classes in the ontology:** domain concepts are represented as classes in the ontology. The vocabulary used in the ontology to characterize the domain should confirm to its usage by the domain experts. Classes are consolidated concepts within the domain.
2. **Arrange the classes in a taxonomic (subclass–superclass) hierarchy:** Find specialization/generalization relations between the concepts of the domain. Organize the domain concepts such that the top most concepts are the most generalized ones, concepts become more specific as we approach the bottom of the hierarchy. The leaf concepts are the most specific ones.
3. **Define object properties (roles):** starting with the top most concepts, identify the properties that hold for these concepts and all subsequent child concepts. Object properties identify the allowed relations between the different concepts of the domain. Each object property has a *domain* and *range*. A domain specifies the concept to which it belongs, while the range specifies the concepts that it is allowed to interact with (i.e. connect to). Define all the concept properties moving from the more generic concepts to the specified ones.
4. **Define data properties:** concepts may also be associated with data properties and describing allowed domains (types) for these properties:

²² Terminology used is based on the Iterative Engineering approach defined in [Noy et al., 2001]

5. **Define additional restrictions:** identify additional restrictions that may be defined to govern the relations between the concepts (classes) (e.g., disjoints, unions, and intersections). Additionally, restrictions can be used to define rules for inferring new knowledge.

Once the ontology is defined (i.e. the vocabulary and structure of the knowledge which it represents) the ontology could be populated with objects (also called instances) that commit to the structure of the ontology. In this case, the ontology is referred to as a *knowledge base*.

4.2.4 Rule-Based Knowledge Representation

Rule-based systems form a different category of knowledge representation mechanisms. Instead of representing knowledge in a declarative static way, rule-based systems represent knowledge in terms of a set of rules which instruct the system on how it should make use of the knowledge or “facts” it stores. A rule-based system consists of a set of IF-THEN rules, a set of facts, and an inference engine. An inference engine is an interpreter controlling how the rules are applied to these facts [Hayes-Roth, 1985]. A typical rule is structured as follows:

Syntax: IF <premise> THEN <action>

The rule premise can consist of a series of clauses and is often referred to as the antecedent. The premise evaluates to a Boolean value. In the premise, the logical connectives AND, OR and NOT can be used. The action refers to a series of statements that hold when the premise is true. The action is also referred to as the consequent. The interpretation of a rule is that if the antecedent can be satisfied the consequent can too. If the consequent defines an action, the effect of satisfying the antecedent is to schedule the action for execution. If the consequent defines a conclusion, the effect is to infer the conclusion [Hayes-Roth, 1985]. For example the rule:

IF Male(x) AND hasChild(x,y) THEN Father(x)

also represented as,

IF Male(x) \wedge hasChild(x,y) \rightarrow Father(x)

This rule states that if some object x is a male and it has a hasChild relationship with another object y then the object x is also a Father. Rules affect the knowledge contained in the system. This knowledge is represented by a set of facts which express assertions about properties, relations, and propositions (e.g., Male(‘peter’)).

Rule-based systems can reason over data in two different ways: forward chaining and backward chaining. In a forward chaining system, the system starts with the initial facts, and keeps using the rules to draw new conclusions (or take certain actions). In a backward chaining system the system starts with some hypothesis (or goal) and tries to prove the correctness (or incorrectness) of this hypothesis. Which type of reasoning mechanism is suitable depends on the problem to be solved. Forward chaining systems are primarily suitable for data-driven problems, while backward chaining systems are more suitable for goal driven problems.

4.3 Semantic Web Knowledge Management Techniques

The Semantic Web is simply a web of data, described and linked in ways to establish context or semantics that adhere to defined grammar and language constructs [Hebeler et al., 2009]. Tim Berners-Lee envisioned the Semantic Web as “*The Semantic Web is not a separate*

Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation.” [Tim Berners-Lee et al., 2001].

The Semantic Web community proposed several knowledge representation mechanisms, each different in its expressivity and thus power and usage. The primary goal of these languages is to provide a standard among all users of the Semantic Web. In the meantime they are light-weighted to meet the not so sophisticated need of reasoning on the Web and yet meet its high availability requirements. In this section, we briefly present the Semantic Web technologies used in this thesis, namely OWL, querying OWL ontologies, and reasoning on OWL ontologies.

4.3.1 OWL

The Web Ontology Language (OWL) extends the RDF Schema²³ and uses the same RDF syntax as its base grammar. Additionally, OWL uses the vocabulary of RDF and RDFS where possible, so RDF²⁴ and RDFS tools could process OWL ontologies that fit into their limited expressive power. In terms of semantics, OWL is heavily based on Description Logic. Furthermore, OWL includes mechanisms to import other ontologies and Semantic Web documents across the Semantic Web.

An OWL ontology consists of a set of axioms and facts that describe the domain. Instead of the typical RDF triple (subject, predicate, object), OWL describes the domain in terms of classes, properties, individuals, data types and values (also called concrete domains in Description Logics). Classes represent concepts in the domain; they can be organized in a taxonomy like structure to indicate sharing of characteristics among the concepts (i.e. generalization-specification relation). A class is described by means of its name. Furthermore, anonymous classes can be described; in this case class descriptions can be composed from all of the above components using various constructors (e.g., union and intersection). Properties describe relationships (also roles) between pairs of individuals. Individuals represent the instances that exist in the domain; an individual can belong to one or more classes. In OWL, individuals can have data type attributes. OWL uses the XML Schema data types, for example: `car1 hasColor "red"`, where "red" is a string.

In the original proposal of OWL (OWL 1, and OWL 1.1), OWL has three increasingly expressive sublanguages: OWL Lite, OWL DL, and OWL Full [OWL Web Ontology Language Overview, 2004]. OWL Lite supports those users primarily needing a classification hierarchy and simple constraints. It has a lower formal complexity than OWL DL. While OWL DL supports those users who want the maximum expressiveness while retaining computational completeness (all conclusions are guaranteed to be computable) and decidability (all computations will finish in finite time). OWL DL includes all OWL language constructs, but they can be used only under certain restrictions (for example, while a class may be a subclass of many classes, a class cannot be an instance of another class). OWL DL has the same expressivity as Description Logics. On the other hand, OWL Full provides maximum expressiveness and the syntactic freedom of RDF with no computational guarantees. This comes on the cost of its reasoning capabilities; therefore reasoning with OWL full is undecidable.

²³ [RDF Primer, 2004]

²⁴ [<http://www.w3.org/TR/rdf-syntax-grammar/>, 2004]

As of November 2009, W3C introduced a new version of OWL, OWL 2 [OWL 2 Web Ontology Language Document Overview, 2009]. OWL 2 is based on SROIQ²⁵(D) and so extends OWL with qualified cardinality restrictions and with significantly extended expressivity with respect to properties. For example, OWL 2 provides the ability to assert that properties are reflexive, irreflexive, asymmetric, and disjoint, and the ability to compose properties into property chains. OWL 2 also weakens the name separation restriction imposed in OWL. In OWL 2 the same name can be used for a class, a property, and an individual. [Horrocks and Patel-Schneider, 2010]. Similar to OWL, OWL 2 has three profiles which define language fragments that have desirable computational properties and in particular lower worst-case complexities for the inference problems related to OWL DL²⁶. OWL 2 profiles are: OWL 2 EL, OWL 2 QL, and OWL 2 RL. OWL 2 EL is based on the EL++, a Description Logic for which standard reasoning problems can be performed in time, that is, polynomial with respect to the size of the ontology. In OWL 2 EL, the restrictions on class expressions rule out the use of universal quantification, cardinality restrictions, disjunction, negation, enumerations involving multiple individuals, and most property characteristics. OWL 2 QL is based on DL-LiteR, a Description Logic for which conjunctive query answering can be implemented using conventional relational database systems and so can be performed in LOGSPACE with respect to the size of the data (individual axioms). It is aimed at applications that use very large volumes of instance data, and where query answering is the most important reasoning task. The OWL 2 RL profile is aimed at applications that require scalable reasoning without sacrificing too much expressive power. It is designed to accommodate both OWL 2 applications that can trade the full expressivity of the language for efficiency, and RDF(S) applications that need some added expressivity from OWL 2. Inspired by Description Logic Programs, OWL 2 RL defines a syntactic subset of OWL 2, which is suitable for implementation using rule based technologies, and presenting a partial axiomatization of the OWL 2 RDF-based semantics in the form of first-order implications that can be used as the basis for such an implementation. [Horrocks and Patel-Schneider, 2010].

4.3.2 Querying RDFs/OWL Ontologies

The W3C proposed a query language for querying RDF graphs, named SPARQL – *Simple Protocol And RDF Query Language*. SPARQL is a standard language for querying RDF data published on the web, either stored natively or viewed via middleware. SPARQL offers a syntactically SQL-like language for querying RDF graphs via pattern matching, as well as a simple communication protocol that can be used by clients for issuing SPARQL queries against RDF graphs [Della Valle and Ceri, 2011]. SPARQL can exploit some Semantic Web inference mechanisms allowing applications to query information from more than one RDF graph at a time or alternatively query integrated information from multiple RDF graphs. For example, it supports queries whose answers are not directly specified in the RDF graph, but that can be inferred using a set of inference rules.

Moreover, given that data can be published on the web using different vocabularies, SPARQL specifications propose different query forms based on the endpoint(s) being queried and based on how the results of the query should be returned. For example, the SELECT and CONSTRUCT forms are suitable for issuing queries against known endpoints that expose data

²⁵ SROIQ is an extension of the description logic underlying OWL-DL, SHOIN, with a number of expressive means to improve its expressivity and which do not affect its decidability [Horrocks et al., 2006].

²⁶ Reasoning with OWL DL is based on an extension of description logic named SHOIN [Horrocks et al., 1999]

using known vocabularies. The SELECT form returns results in a tabular format using XML, whereas the CONSTRUCT form returns results in RDF. Therefore allowing users and applications to query the knowledge represented with one or more RDF graphs.

4.3.3 Reasoning on RDFs/OWL Ontologies

Reasoning means using the already existing knowledge (axioms and assertions) to infer new knowledge. A program that does so is called a reasoner or an inference engine. The RDFS and OWL standards define what inferences are valid, given certain patterns of triples. The semantics of the vocabulary of RDFS and OWL instruct inference engines to how they should infer given patterns. As an example the `rdfs:SubclassOf` defines hierarchies of concepts, therefore any valid RDFS/OWL inference engine should infer the complete hierarchies of concepts within the ontology. For example given the two triples `(Female, rdfs:SubclassOf, Person)` and `(Mother, rdfs:SubclassOf, Female)`, a valid reasoner should infer the triple `(Mother, rdfs:SubclassOf, Person)`.

Most reasoners on the Semantic Web are Description Logic reasoners, which apply the Tableau Reasoning algorithm [Baader et al., 2003]. Some of the well-known Semantic Web reasoners are: FaCT++ [Tsarkov and Horrocks, 2006], HermiT [Motik et al., 2007], Racer pro [Haarslev and Möller, 2001], and Pellet [Sirin et al., 2007].

FaCT++ is an open source C++ reasoner that implements the tableaux algorithms. The reasoner performs classification²⁷ of the ontology, while doing so it uses a KB satisfiability checker in order to decide subsumption problems for given pairs of concepts. A drawback of FaCT++ is that it does not take OWL ontologies directly nor any remote files. A utility program digFaCT++ takes local files in DIG²⁸ and translates it to the reasoner through the DIG interface. The *tell* and *ask* commands are used to communicate with the reasoner.

HermiT is a new OWL reasoner based on a novel “hyper-tableau” calculus. The new calculus addresses performance problems due to non-determinism and large model size. Similar to Fact++, HermiT can determine whether the ontology is consistent, and identify subsumption relationships. It can handle multiple OWL ontology formats, and supports both OWL DL and OWL 2. Additionally, HermiT supports DL-safe SWRL rules. HermiT is available as an open-source Java library, and includes both a Java API and a simple command-line interface. Being quite recent HermiT is not very stable yet.

Racer is a commercial OWL reasoner, which implements a highly optimized tableaux calculus for deciding the ABox consistency. It supports highly optimized special purpose inference procedures for sublanguages of Description Logic, which are applied automatically whenever applicable to the input problem for maximal performance. The sound and complete inference algorithm with the highest performance is selected automatically.

Pellet is an OWL DL and OWL 2 reasoner; it implements the Tableau Reasoning algorithm. Additionally, Pellet provides some support for ontology debugging by providing support for justifying entailments. Furthermore, it provides support to reason over ontologies containing SWRL rules [Horrocks et al., 2004]. It provides support for safe SWRL rules, i.e. rules that would keep the reasoning process over the ontology sound and complete. Furthermore, Pellet provides support for some of the SWRL built-ins. Pellet was implemented with usage for the Semantic Web in mind; therefore it provides some important facilities such

²⁷ i.e. computes and caches the subsumption partial ordering (taxonomy) of named concepts

²⁸ The DIG Interface is a standardised XML interface to Description Logics systems developed by the DL Implementation Group

as reasoning on individuals (ABOX reasoning) and query answering. Pellet is open source and is available as an open-source Java library, and includes both a Java API and a simple command-line interface.

4.4 Knowledge Management Applied to Software Variability

Many authors have noticed the importance of applying knowledge management principles to analyse and understand software variability or to aid in the development of variable software. Two principles exist. Firstly, there is the use of different knowledge representation techniques to map knowledge in variability models, namely feature models. The mapping is done such that every “feature” in the feature model is considered a first class concept in the defined knowledge model. Variability represented by these features is mapped to restrictions that are made part of the knowledge model. These restrictions define how the different concepts (i.e. features) are linked together via the relations. The type of the concepts that the feature maps to depends on the knowledge representation technique used. For example, some of the existing mappings are implemented via Description Logic (DL), Higher Order Logic (HOL), and OWL. Examples of the first principle have been shown in section 3.1. The second principle for applying knowledge management to analyse and understand software variability is based on using knowledge representation techniques to formulate knowledge within a certain domain (i.e. problem domain), in addition to the variability within that domain. In a next step, this information is used when developing the software. Examples of the second principle are:

Mohan & Ramesh [2003] define an ontology that catalogues the different concepts associated with variability in product line development, such as variation points, variants, variability phase, and variability patterns. The ontology is then used to define the elements characterizing the knowledge elements necessary for managing variability in product lines. The defined ontology also captures various variability modelling mechanisms thereby aiming to provide support for mechanism selection. The authors also provide a knowledge management tool integrated with the ontology to facilitate knowledge capturing and retrieval for variability management. The developed system was based on Microsoft Access.

Lee et al. [2007] use ontology similarity measure to analyse feature models. A semantic-based analysis criterion is proposed to analyse commonality and variability of features by changing a feature model of a specific domain to a corresponding feature ontology. The purpose of the approach is to overcome feature ambiguity problems (e.g., duplication of features and inaccurate meaning of terms used) when multiple stakeholders are modelling the system. The approach starts with defining a syntactic meta feature model and attributes of each element in the model. Next, a feature model of the target domain is constructed based on the defined meta feature model. The constructed model is then transforming into an ontology and store it in a Meta Feature-ontology Repository. Next, a feature model of the same product line is constructed and transformed into an ontology. Ontology based semantic similarity measures are then used to compare the two ontologies for their similarity and differences. Common features represent common domain concepts, which could be reused in other product lines while variable concepts are inspected to check whether they actually represent new concepts, or not. New concepts are then defined as variable characteristics of the product.

Ferreira et al. [2009] propose a formalization of an approach that combines multi-stage (time-variant stages), with ontological support and multilevel primitives (abstraction levels) for the insurance domain software process development. They propose that conceptual models underlying the different business domains (like banking, insurance, industry, and others) need to be explicitly defined by ontologies to promote a shared understanding of these concepts.

Ontologies will act as a guideline containing the core business and development concepts required by the model driven tools to generate specialized and business validated software artefacts.

Johansen et al. [2010] motivate the need to bring feature modelling and ontology techniques together to gain the benefits of the formality of ontologies in designing feature models. They propose using an ontology to accurately express the domain of interest. Therefore the authors propose establishing a mapping between the feature model of a given software product line and the ontology that defines the same software product line in order to provide unambiguous semantics of the terms used in the feature model. Additionally, the authors propose that a mapping should also be defined from the ontology to feature models to allow processing an ontology and reverting the information back to a feature model.

4.5 Summary

In this chapter, we have given an overview of the domain of knowledge management. Knowledge management is the discipline under which knowledge in a certain domain is made explicitly available for the sake of designing processable models. Processable models make knowledge about a certain domain explicit and turn it to actionable knowledge, i.e. knowledge that is *available* and *computable*. We also presented some of the basic concepts for defining a conceptual model of some domain of interest. Next, we listed some of the relevant knowledge representation techniques which allow to formally represent the concepts, roles and restrictions to model a certain domain of interest. Next, we briefly presented different data representation technologies that are used in this thesis. We presented how the Semantic Web provides the tools that make knowledge machine processable and provides standardized languages for doing so. Additionally, it provides the technology that allows to reason on this information as well as querying this information. Finally, we conclude with related work in the area of using knowledge management techniques to represent and realize software variability.

Chapter 5

Challenges for Software Variability Modelling

The purpose of this chapter is to present the conclusions drawn from our study on mainstream feature modelling techniques with respect to the challenges and limitations of current feature modelling techniques. Identifying these was the first step towards identifying the requirements for effective feature modelling and management of the related knowledge. Overcoming them is a major objective of our proposed solution, as already stated in section 1.6.

As already explained, feature modelling is an important phase for efficiently planning the variability opportunities of the software (or software product line) under consideration. As mentioned, feature-oriented variability modelling is a preferred technique to discover and represent variability, because features are abstractions that all stakeholders can understand. Furthermore, features represent the building blocks that makeup the software. The feature modelling technique shows how these building blocks could be varied to provide the necessary required differentiation between the possible products. It also shows the blocks that remain constant to every product of the product line. As such, feature models contain important knowledge that is not only useful for planning the variability opportunities of the software but also in later stages of its life cycle. Feature models allow companies to be in control of the complexity of their products because it contains the information about the features of the products, their variability, and complexity. This means that at any point in time, the company should be able to query and inspect the model(s) for information (i.e. management of the feature models' information), therefore allowing for more flexibility in case of changes, or when introducing new features or new products to the product line.

The challenges presented in this chapter were the driving force for the Feature Assembly Modelling Method (presented in chapters 6 and 7) and the Feature Assembly Reuse Framework (presented in chapter 9) presented in this thesis. The management of information in feature models is handled in chapter 10. In this chapter, we focus on identifying the requirements for effective feature modelling and information management. First, in order to answer research questions RQ1.1 (*Do current feature modelling techniques provide means to understand and express complexity?*) and RQ1.2 (*What are the limitations and practical issues of current mainstream feature modelling techniques?*), we will discuss the limitations of mainstream feature modelling techniques (described in section 5.1), and secondly, in order to find an answer research question RQ2 (*How can the knowledge in feature models and features be captured and unlocked?*), we will identify the challenges related to managing the information provided by feature models (described in section 5.2). Based on this information, we provide a list of requirements for Feature Assembly in section 5.3.

5.1 Limitations of Mainstream Feature Modelling Techniques

As already mentioned in chapter 2, feature models are used to represent the commonalities and differences in variable software by describing the features that make up the software, how they are related, and how they contribute to the variability of the software. Feature models relate features by means of a AND/OR hierarchical structure, describing how features are broken up into more finer-grained ones. There are no clear guidelines on how this decomposition should be done and when it should stop; it is left to the intuition of the practitioner. For small applications this works fine, as features are perceived quite easily and often represent the main system capabilities and components. Yet, in practical cases, there is usually great doubt on how to apply the feature modelling technique. As a consequence, companies have defined their own notations and techniques to represent and implement variability. Examples are Bosch [MacGregor, Bosch Experience report, n.d.], Philips Medical Systems [Jaring et al., 2004] and Nokia [Maccari et al., 2005]. Yet, the proposed notations are tailored to each company's specific needs for modelling variability in their product line. Bosch adopted a hierarchical structure of features similar to feature models but new semantics were introduced to better indicate how features relate to variability and how they relate to each other. For example, an "is realized by" relationship was introduced to represent how features depend on each other. Philips Medical Systems had a scalability issue; therefore, they defined the "Variability Categorization and Classification Model" which helped them use a building block method to define variability in their MRI product line. While feature interaction and scalability issues were more important for Nokia, they adopted a separation of concerns approach for devising higher-level features. For them, the evolution and changing of features over time was very important, documentation was used to specify the system features and relations. Moreover, and a confirmation of our findings, a recent study on the application of feature models in practice [Hubaux et al., 2010b] reveals that there are very few reports on the use of feature models in practice. That study shows that out of the available literature of software variability only 16 cases were relevant. Furthermore, this study shows that only two of the 16 cases claim success in applying feature models, while two reports mention a failure in using the technique. In addition, five cases were false positives i.e., cases for which the applications of feature models turned out to be missing or too vague to tell anything about their fitness.

We started with analysing the existing mainstream feature modelling techniques in order to understand their capabilities²⁹ in addition to their limitations. The following limitations were identified:

- L1. Difficulties in using the feature modelling technique in practice**
- L2. Ambiguity in modelling concepts**
- L3. Lack of abstraction mechanisms**
- L4. Limited reuse opportunities**

Limitations L3 answer our research question RQ1.1 questioning about the support for understanding and expressing complexity. While limitations L1, L2, and L4 partially answer our research question RQ1.2 looking for the limitations of current feature modelling techniques. Limitation L4 also provides some insight on the obstacles of introducing modelling with reuse for creating feature models; this will help us answer RQ1.5 (How can the principle of "modelling with reuse" be introduced to feature modelling?). We will discuss each of these

²⁹ More on this analysis will be given in chapter 6; we focus in this chapter on the limitations of mainstream feature modelling.

limitations in more details in the next subsections. We also list the impact of these limitations and the consequences we took in order not to fall in the same pitfalls.

5.1.1 Difficulties in Using the Feature Modelling Technique in Practice

As already mentioned in chapter 2, there are many variations of the original feature modelling technique, FODA. Some of these variations have different notations to represent the semantics provided by FODA such as FORM [Kang et al., 1998] and FeaturSEB [Griss et al., 1998]. While other feature modelling techniques define new notations as well as new semantics for modelling the variability of features, such as Riebisch et al.'s feature model [Riebisch et al., 2002], and PLUSS [Eriksson et al., 2005]. Other extensions of FODA, such as extensions to include cardinality [Czarnecki et Kim, 2005] and feature constraints [Ye et Liu, 2005], also exist. With all these differences in semantics, as well as in notations, it is not obvious for practitioners to decide which one is the most appropriate to select. In addition to the existence of many feature modelling methods, the FODA based feature modelling techniques adopted in these methods also pose some other problems that hinder their use in practice. We list here these issues and their consequences.

L1.1. What is a “feature”?

Many definitions of “*feature*” exist; actually each technique is using its own definition. For example, some of the common definitions of feature are:

1. A feature is a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems [Kang et al., 1990].
2. A feature is a logical unit of behaviour specified by a set of functional and non-functional requirements [Bosch, 2000].
3. A feature is an increment in program functionality [Batory, 2005].
4. A feature is a functional requirement, a reusable product line requirement or characteristic [Bosch, 2000].
5. Features are prominent and distinctive system requirements or characteristics that are visible to various stakeholders in a product line [Lee et al., 2002].
6. A feature is a requirement or characteristic that is provided by one or more members of the software product line [Gomaa, 2005].
7. A feature is a small client valued function [Palmer and Felsing, 2001].
8. Features are an interpretation of the requirements [Van Gorp et al., 2001].
9. A feature is a triplet, $f = (R;W; S)$, where R represents the requirements the feature satisfies, W the assumptions the feature takes about its environment and S its specification [Classen et al., 2008].

Consequences

- It can be seen from these different definitions that features can be considered from different points of view. While the first, fifth and seventh definition take the user's perspective for defining what a feature is, the second, fourth, sixth, eighth, and ninth definition take the requirements perspective for defining what a feature is. Additionally, the third and the seventh definition take the functional perspective for defining what a feature is. This observation implies that a feature may have many different faces, making it *difficult to base feature analysis on only one point of view* or

one perspective. Therefore, when analysing features of software, many perspectives could be taken into consideration.

L1.2. No clear modelling guidelines.

Feature modelling is not widely adopted in industry. This may be due to the fact that the modelling concepts are not well defined, as well as to the lack of practical guidelines for creating feature models [Lee et al., 2002 b]. As stated by Lee “the fuzzy nature of feature makes it difficult to formalize its precise semantics”. Furthermore, no clear guidelines exist on the level of detail needed for a feature model be, i.e. when should the decomposition of features stop. It is implicit in the feature model literature that the feature models provide an overview of the top-level features visible to users. In our opinion, the modelling process should stop when there is no longer any variability in the modelled features. If a node (or possible decompositions) does not introduce variability then the modelling should stop, this is because feature models are intended to model variability in the product line and not intended to model functionality per se. On the other hand, the goal of feature modelling is modelling variability and commonality. Therefore, practitioners should focus on identifying characteristics of products in terms of commonality and variability, rather than describing all details of the product implementation, which could be done using other modelling techniques.

Consequences

- There is a need for a method that guides the users in how to use the modelling technique. For example, *it should be easy to know what is a feature and what not*, and when the modelling process should stop, i.e. a criterion for stopping feature decomposition should be defined.

L1.3. Missing Link with Variability Specifications

Mainstream feature modelling techniques don't link their notation of features with the notation of *variation point* and *variant*, which are preferred terms among stakeholders interested merely in variability [Bosch, 2001]. Such a link would facilitate communication between feature modelling practitioners and other stakeholders interested merely in variability.

Consequences

- It is important to provide *a mapping from variability notations* (in terms of *variation points* and *variants*) *to feature model notations* (in terms of features, feature types and feature relations).

L1.4. Hierarchical top down modelling approach.

Humans don't immediately view concepts in terms of hierarchical structures, rather they use hierarchical structures to analyse and organize the knowledge they have gained about a certain domain but not for acquiring this knowledge [Aamodt, 1995] [Anderson, 1996]. Moreover, the hierarchical top down decomposition structure adopted in feature models makes maintenance difficult. This is due to the significant amount of changes required when new concepts are introduced, or when existing concepts need to be removed.

Consequences

- There is a need for a modelling approach that allows *combining bottom up and top down approaches to define features*.
- There is a need for a modelling technique that *easily support introducing new features or new variations of features* and changes.

5.1.2 Ambiguity in Modelling Concepts

The modelling concepts defined in subsequent FODA based feature models add a great deal of power for the modelling language in order to achieve more flexibility. Yet, in order to do so, they also add a great deal of complexity for the modelling language. This flexibility in the language constructs became a double edged weapon causing practitioners to have some doubts on applying the technique, and may very easily result in models of questionable quality. We list here some of the ambiguity problems caused by the non-rigid modelling concepts of mainstream feature modelling and their consequences.

L2.1. Feature decomposition is overloaded

The decomposition mechanism used in feature models is based on functionality, i.e. a whole-part decomposition, as well as on variability, i.e. a generalization–specification decomposition. Not having a clear distinction between these two fundamentally different types of relations, i.e. functional decomposition and variability decomposition, makes the modelling process difficult and is a source of errors [von der Massen and Lichter, 2004] [Riebisch, 2003]. It also makes the resulted feature models difficult to understand for non-domain experts, while one of the purposes of the use of feature models is to communicate the variability opportunities in the product line (or domain) to the different stakeholders (e.g., marketing specialists, architecture engineers, developers, innovation specialists, or even valued customers). Overloaded decomposition of features will make feature models confusing and the information will not rigorously be conveyed to the involved stakeholders. This is because feature models do not make an explicit distinction between composition and specialization. This may introduce problems. For example, figure 5.1 shows the Graph product Line (GPL) problem introduced by [Lopez-Herrejon and Batory, 2001]. The figure represents a Graph Product Line that is optionally composed of a *Search* feature and mandatory composed of a *Graph Type* feature. The *Search* feature has two alternatives: DFS and BFS. While for the *Graph Type* feature two sets of alternative groups are identified. The first group has two alternatives: *Directed* and *Undirected*, while the second group has two alternatives: *Weighted* and *Unweighted*.

This introduces two problems. Firstly, the model holds implicit information (by not naming the two concepts for which the two sets provide alternatives) leaving it to the intuition of the user to understand that there are two concepts that makeup graph type (i.e. direction and weight). Secondly, *Graph Type* is a mandatory feature, while its successors are alternative features. Therefore, it is not clear whether at least one feature of one alternative group should be selected, or one feature of each group should be selected. The first part of the

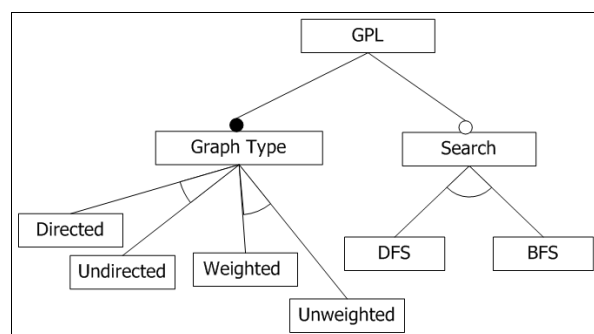


Figure 5.1 Feature Model of GPL, ambiguity in Graph Type definition

problem is related to the fact that the Graph Type feature is *functionally* decomposed in two sub concepts ‘direction’ and ‘weight’, which are implicitly represented by the alternative branches: *Directed* and *Undirected*; *Weighted* and *Unweighted*, which indicates two variability decompositions. The second part of the problem is a result of not forcing the modellers to represent the implicit feature decomposition information, and allowing to specify functional decomposition in combination with variability decomposition at the same time.

Consequences

- There is a need for a modelling technique that supports the *distinction between function decomposition and variability decomposition*.

L2.2. Redundancy within the Feature Models

Figure 5.2 shows another example of ambiguity that is caused by redundancy resulted from combining different types of variability: F is optionally composed of F1, and at the same time F1, F2, F3 and F4 are alternative descendant features of F. Although this ambiguity can be resolved by normalizing (see section 2.3.1), i.e. allowing each feature to have only one type as shown in the figure, the modelling method itself does not prevent such situations.

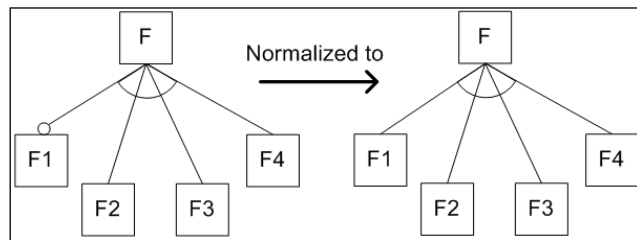


Figure 5.2 Example showing ambiguity of feature models

Consequences

There is a need for *rigorously defined modelling concepts*, and *methods that control redundancy* in feature and variability specifications.

5.1.3 Limited Reuse Opportunities

In current feature models, a feature is given a type that indicates how the feature contributes to the variability of the system. This limits the possibilities to reuse a feature in a different context. For example, a *bank transfer payment* feature may be mandatory in one setting while optional in another (e.g., depending on the target market or country). As the type (here mandatory or optional) is inextricably associated with the feature, it will not be possible to reuse the feature as it is. In addition, change is also an issue. It is quite difficult to add new features or change an existing feature (e.g., change its variability type). For example, a *Payment* feature may have two alternatives *Bank Transfer* and *PayPal* (alternative features), when targeting new markets this feature may need to be extended with other payment methods (e.g., *Visa*, *Mastercard*, and *Bancontact/Mister Cash*). Furthermore, suppose that the *Bank Transfer* feature needs to become mandatory to suit all markets while there is a need to select one or more of the other payment features (OR Features). As the type *mandatory* is inextricably associated with the feature, it is not possible to reuse the feature as it is. Such a change requires deleting the old Alternative Feature group, creating a new OR group, and changing the type of the *Bank Transfer* feature to mandatory, this process is shown in figure 5.3. Note that adding

and removing branches in the feature model tree may not always be a straightforward task (e.g., it may need backtracking and reconstruction of more than one branch or even level).

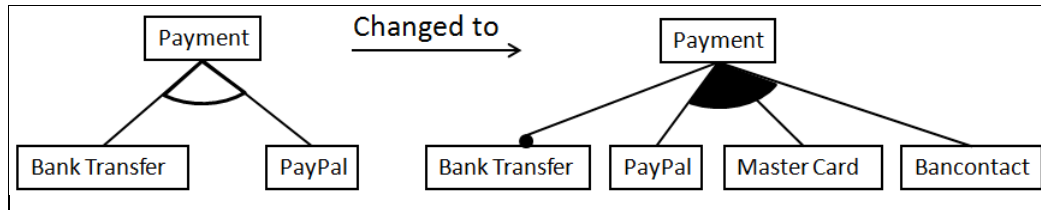


Figure 5.3 Example showing the impact of change in Feature Models

Consequences

- There is a need for *separating the feature from how it contributes to variability*; it must be possible to use the same feature in different variability specifications.

5.1.4 Lack of Abstraction Mechanisms

Designing large and complex systems is not easy. Dealing with different aspects of a system at the same time is difficult and error prone. Therefore, there is a need for abstraction mechanisms that allow modelling large and complex systems by scaling down the complexity and size. FODA and subsequent FODA based feature modelling techniques lack explicit abstraction mechanisms. Usually, high-level features are decomposed into lower level features in the feature model, but in systems with large feature interactions or a large number of features this is a cumbersome task. A successor of FODA, FORM, defined four categories to which features of the system belong: *operating environments*, *capabilities*, *domain technology*, and *implementation techniques*. The capabilities category is further categorized into *functional features*, *operational features*, and *presentation features*. However, we see this categorisation process as very fragile and impractical. First of all, it is not always easy to decide on the category of a feature. Secondly it is not a true abstraction mechanism but rather a grouping mechanism. In reality, a feature may have many faces, which make categorizing features into a single category a difficult task. Thirdly, the proposed categories will not be suitable for all types of application.

Consequences

- There is a need for *abstraction mechanisms* that allow dealing with complex and large systems.

5.2 Challenges in Managing the Information in Feature Models

This section aims identifying the required input for answering our research question RQ2, which deals with capturing and unlocking the knowledge contained in feature models. There is a need to understand the added value. Additionally, there is a need to determine how to support unlocking and sharing of information. The need and difficulty in managing variability and commonality information comes from the fact that software systems have grown in terms of the number of features they hold and the complexity of relations and dependencies between these features. Therefore the created feature models could become very large due to the increasing number of features ranging from a few hundred and jumping up to a few thousand

[Bosch, 2005]. This makes the process of managing the different relations between the different features of the system, a very difficult task if done manually. Furthermore, there is a need for supporting the correctness and validity checking of the created feature models, i.e. detecting conflicting or semantically incorrect feature relations. There is a need for managing the information in feature models both at modelling time and afterwards, each phase with a slightly different focus of information management. We will describe each in more details.

5.2.1 Information Management to Support Feature Modelling

Feature modelling is an iterative process involving many stakeholders. Today, the development of software is usually distributed over several teams. As a result, feature modelling is usually a distributed task. This means that the management of features becomes more complex. Two forms of distribution exist. Firstly, distribution in terms of functionality, i.e. different feature models may be created for different parts of the system. Secondly, distribution in terms of location due to the fact that different people located at different places may be involved in the process, each creating parts of the models. Usually the distributed created parts are not independent; therefore there is a need to allow linking features from different parts. Additionally, feature dependencies may exist between different parts of the system. There is a need for maintaining these dependencies by ensuring that the segments of feature models contain no conflicting information. Some software organizations tend to deal with this issue by trying to come up with a distribution that reduces the dependencies. For example, in a report [Maccari and Heie, 2005], Nokia is stating: “*The heuristic we adopt is ‘avoid dependencies between people that are not located under the same roof and under the same organizational entity’*”. However, it is obvious that complete independency is not possible.

Furthermore, dependencies between features are poorly documented. Although, dependencies will greatly influence the capabilities of the software (through influencing the selection of features) and the anticipated functionalities, most of the reasons for having these dependencies and relations cannot be easily captured by the different feature models [Zave, 2004]. Feature dependencies define how features interact together to meet the system’s specifications. Van Gurp et al. [2000] state “It is virtually impossible to give a complete specification of a system using features because the features cannot be considered independently”. According to Griss [2000] the feature interaction problem is characterized as follows: “The problem is that individual features do not typically trace directly to an individual component or cluster of components - this means, as a product is defined by selecting a group of features, a carefully coordinated and complicated mixture of parts of different components are involved”. Therefore, there is a need for a richer set of feature dependencies that allow expressing the different semantics (i.e. reasons) of the dependencies. Although this richer set will not add more expressivity from a configuration point of view, it could convey the rationale of the dependencies to the stakeholders involved in the modelling process (e.g. customers, marketing and sales, innovation researchers). This kind of information about feature dependencies should be explicit in the model.

While the modelling method should provide abstraction mechanisms that facilitate the modelling process, there should also be abstraction mechanisms that allow modellers and stakeholders of one subsystem to inspect the feature models of other existing subsystems, while controlling the desired level of details in viewing this information.

Consequences

- There is a need for a *processable knowledge model for feature models* that allows users to collaborate and share information contained in their feature models

- The processable knowledge model should make it possible to *support different abstraction mechanisms to view information*.
- There is a need for expressive feature dependencies in which the rationale (i.e. the reason for why the dependency holds, e.g. domain constraint, product line scoping, technical constraint ... etc.) of the dependency is not lost.

5.2.2 Information Management of Feature Models

Feature models should allow finding relevant information about the SPL at any stage in the development cycle. Keeping track of the features and their variability and commonality information within an organization helps creating an understanding of the current as well as future business opportunities. It is also important that this information is transparent to all the involved stakeholders at any point in time. Because features are mapped to software assets (e.g. classes, components, libraries, etc.), the influence of the information contained in the feature models goes beyond the domain analysis stage, it has an impact on all later stages in both domain engineering and application engineering.

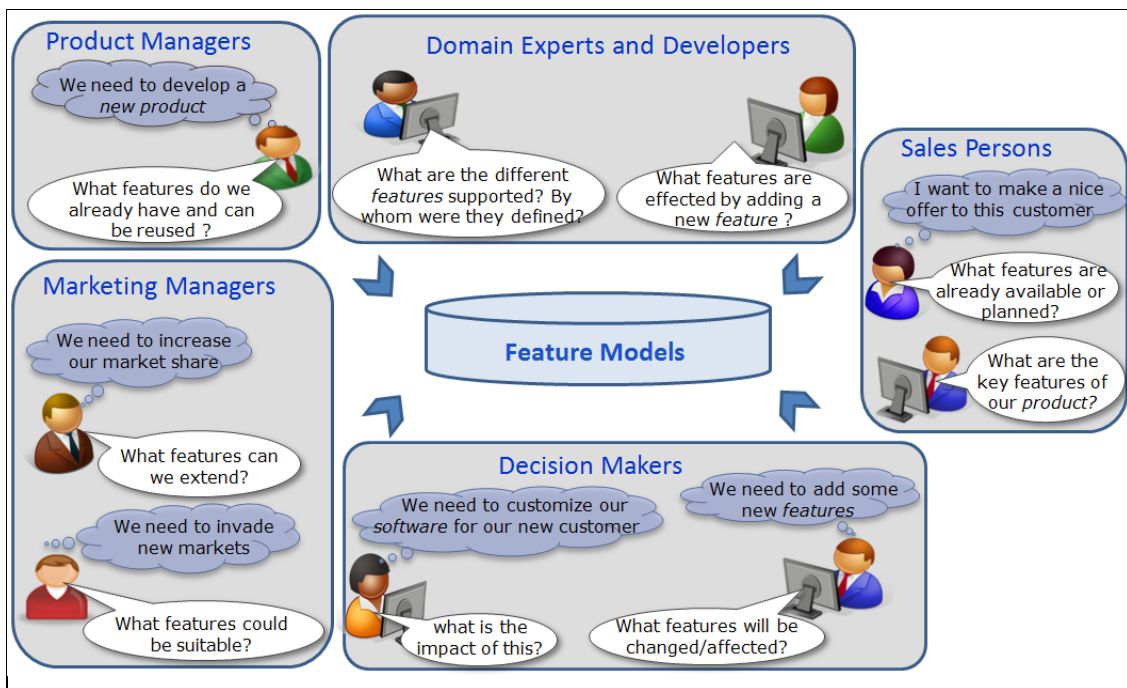


Figure 5.4 Example showing possible use cases for different stakeholders once a knowledge base for the product feature models exist

Figure 5.4 shows some possible scenarios for different types of stakeholders for interacting with the information contained in feature models. These scenarios show that inspecting information contained in feature models and combining it with the expertise and knowledge of the involved stakeholders should allow them to make more accurate decisions. Moreover, business opportunities could be missed due to lack of information about possible or potential variability opportunities. Below, we list the advantages of providing information management for feature models.

1. **Unlock Variability Information:** Variability knowledge about software in companies is very often stored in the heads of people, in paper documents, or at the best documented in the code. This makes it difficult to efficiently retrieve and share this knowledge. There is a need for unlocking and sharing information about product features and their variability in the organization in order to address decisions about future directions or study impact of changes. By doing so, companies could gain the following merits:
 - Expansion of current business opportunities: Companies can identify the similarity between the new required features and the existing ones. This allows identifying possible reuse opportunities or the expansion of current variable features (by introducing new variants).
 - Help understand the impact of new upcoming features: By unlocking information about feature dependencies and features, decisions about future directions or impact of changes could be more accurate.
2. **Enhance Collaboration Among Different Stakeholders:** Collaboration among different stakeholders could be enhanced by enabling efficient retrieval of information about the different existing features and their interactions (i.e. dependencies and relations). Furthermore, with the help of information processing techniques it should be possible to query the (processable) feature model for useful information, such as *possible variants of a certain feature*, *features that exclude another feature*, search whether *a specific feature exists*, and so on. There is also a need to provide different stakeholders with different levels of abstractions. Not all stakeholders are interested in all features of the system; likewise different stakeholders need to be able to explore the system with different level of details (e.g., the level of details required for sales persons is different than that required for developers).

Consequences

- There is a need for a processable knowledge base of feature models, and users should be able to *query and interactively explore this knowledge base*.

5.3 Recommendations for Feature Assembly

In the previous sections, we have showed the limitations of current feature modelling techniques and the consequences they have on the practical use of these techniques. As already mentioned, these limitations have led to defining the Feature Assembly approach, the major contribution of this thesis. Furthermore, we also have indicated the need for a proper feature information management. In this section, we list the requirements formulated for enhanced feature modelling and for efficient feature information management.

Requirements for enhanced feature modelling:

1. Provides unambiguous modelling concepts with clear semantics. In particular, separates the feature from how it contributes to variability; it must be possible to use the same feature in different variability specifications.
2. Provides a rigorous methodology for feature modelling.
3. Provides different abstraction mechanisms to deal with complex and large systems.

Requirements for feature information management:

1. Allow users to collaborate in developing feature models and share information about feature models, as well as, allow users to query for information contained in feature models.
2. Make it possible to support different abstraction mechanisms to view information about feature models.

5.4 Summary

In this chapter, we presented our knowledge acquisition phase in which we explored the limitations of current feature modelling techniques. For the existing feature modelling methods the following limitations were identified: 1) difficulties in using the feature modelling technique in practice, 2) ambiguity in modelling concepts, 3) lack of abstraction mechanisms, 4) limited reuse opportunities. Furthermore, we identified how these limitations can be overcome. We also explored the needs of efficient knowledge management for feature models. We identified the current challenges in efficiently managing information contained in feature models both during the actual feature modelling phase and further on during the product development process and after the development of the product. We identified several needs for such an information management process. There is a need to unlocking information contained in feature models such as feature interactions, crosscutting features between different subsystems or components, and the need for supporting different abstraction levels. Additionally, once feature models are established there is a continuous need to benefit from the information contained in these models whether for maintenance purposes or for future expansion of the software. We concluded this chapter with a set of requirements for an efficient feature modelling method and a processable feature modelling knowledge base.

Chapter 6

The Feature Assembly Modelling Technique

The previous chapter introduced the shortcomings of current feature modelling techniques and pointed out how these shortcomings have affected the efficiency and ease of modelling as well as the quality of the created models. We have also listed a set of requirements that need to be considered by feature modelling techniques in order to overcome the previously mentioned limitations. This should allow the creation of **well-structured**, **scalable**, and **reusable** feature models. Taking into account these recommendations has resulted in the *Feature Assembly Modelling* technique (FAM). Feature Assembly Modelling proposes a limited set of easy to use concepts to model variability and commonality. In the same time, the notations for the concepts restrict the users to creating only well-structured feature models. Scalability is handled via introducing an abstraction mechanism that allows for separation of concerns while doing the modelling. FAM also promotes reusability by supporting both feature reusability and design reusability; this will be presented in chapter 8.

6.1 Feature Assembly Overview

Modelling of variable software is not an easy task due to the fact that variability comes at the price of increasing complexity. Therefore, the variability modelling technique should allow to explicitly represent the variability, clearly indicating its influence on the complexity. The variability modelling technique should aid the modeller in clearly indicating the features that have a variable nature (and therefore contribute to the variability of the overall system) and how they relate to other features in the application. The variability modelling technique should make it easy to spot variability opportunities as well as indicate how these opportunities can be tuned to deliver different products. The technique should also take into consideration that variability highly varies in both its complexity level and granularity levels; the problem becomes more difficult in large software. Variability in software is often identified in terms of the variable characteristics, capabilities, and functionality of the software.

The Feature Assembly Modelling technique (FAM) is a feature oriented conceptual modelling technique for analysing and modelling software that contains variability (aka software product lines). FAM supports identifying the basic building blocks of the software, namely *features* starting at different granularity levels. FAM not only allows identifying features but also how they are related in terms of their compositional structure, their contribution to the variability of the software, and their feature-to-feature interactions.

Feature Assembly Modelling proposes a *perspective*-based abstraction mechanism to deal with large and complex systems. The perspective-based approach utilizes the separation of concerns principle while modelling. This is done by adopting a specific mind-set or point of view when identifying and modelling features. As such allowing to define the features that are relevant from a particular aspect or viewpoint rather than considering all aspects of the whole system at once, which usually results in badly structured models.

The Feature Assembly Modelling process starts with a variability feature analysis phase as shown in figure 6.1 (this will be discussed in section 6.3), which anticipates potential features that make up the software and the level of variability they hold. Next, a set of perspectives that represents the different viewpoints for modelling the software are defined. Each of these perspectives represents a consolidated point of view for modelling the software. We will discuss the perspective-based approach and give examples of the most common set of possible perspectives in section 6.4. The next step is to model the features of each perspective by defining the feature relations and the feature dependencies they hold. This is done using the modelling primitives offered by the Feature Assembly modelling technique. Finally, it is

important to link the separate perspectives (this will be discussed in section 6.5). We start the rest of this chapter by introducing a running example (section 6.2).

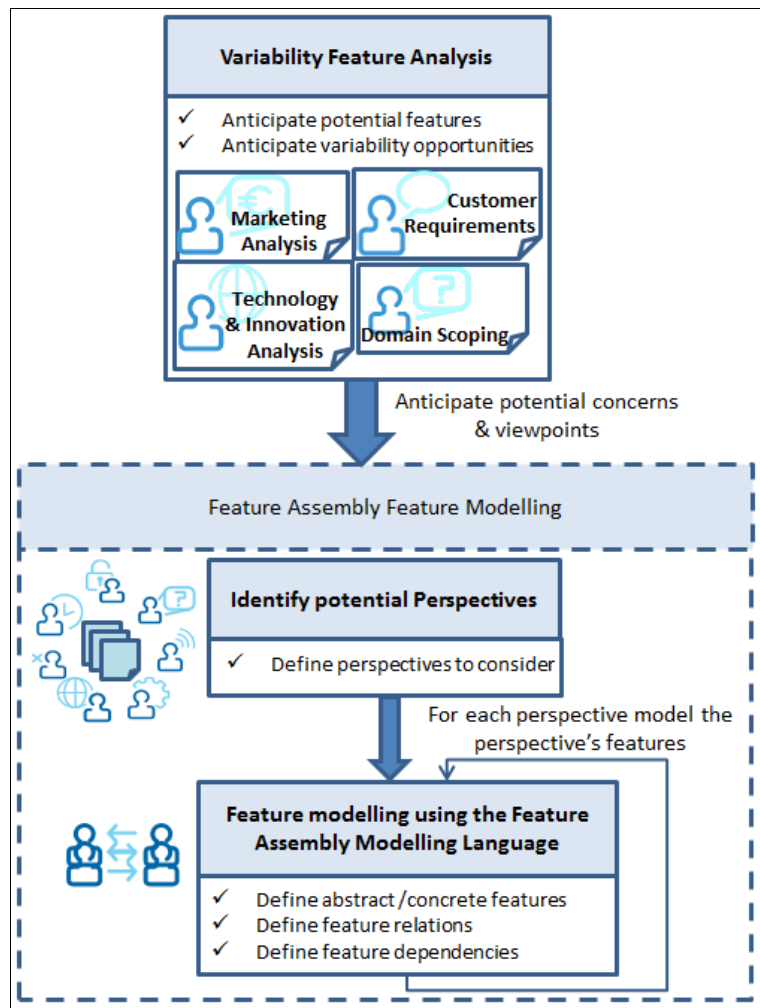


Figure 6.1: Overview of the Feature Assembly Modelling process

6.2 Running Example – E-Shop Product Line

From this point forward, we will be using the running example of an E-Shop³⁰ product line. The E-Shop mainly supports navigating through the product catalogue, ordering of products and shipment of the products to the customers. A common scenario to how this is done is as follows: a customer selects some products and adds them to this shopping basket. He then performs the purchase process and the system verifies and processes the customer's order and issues the payment of the order. Different possibilities for the payment exist. When the order is validated by the system and the payment is accepted, the order is fulfilled and products are delivered to the customer. Products are delivered via electronic delivery or via shipping the physical products to the customer. To support this general scenario, the system should first display the products with their available quantities to the customers. The system should also

³⁰ This case study has been used before as a benchmark in SPL research, a complete feature model of the E-Shop case study can be found at <http://www.hats-project.eu/node/206>

provide facilities for the customers to create an account, manage their account, and do online shopping.

6.3 Variability Analysis

Feature Modelling is a practice to understand the variability of the system being modelled (e.g., a product line). In domain analysis, features refer to problem space concepts that define the characteristics, i.e. functionalities and capabilities, of the desired software. Features are therefore a combination of both *what the software can do*, i.e. functionality and *what the software embraces*; i.e. capabilities. The rules that govern how features could be composed conceive the amount of allowed variability within the software. A broad definition for feature is:

Definition 6.1 (*Feature*)

- ***A feature is a logical or physical unit of the system.***

Note that the definition of feature will be *refined* later on when we discuss perspectives (section 6.4).

When these *units* embrace resemblance, this resemblance is an indication of variability. A variable feature is a feature that has several different forms. A feature may not be variable by itself but contributes to variability by means of being an *optional* part of the product line, in the sense that it is not required to exist in all the products possibly derived from the product line.

Features stem from two main sources, requirements and domain analysis. The first source of features, requirements, is defined by Van Gorp et al. [2001] “Features are an interpretation of the requirements”. In requirements engineering, usually two types of requirements are distinguished, functional requirements and non-functional requirements. The functional requirements describe what functionalities the software must provide. Therefore, they define the functional characteristics of the system and are greatly triggered by the business requirements and domain requirements. Functional requirements define the scope of the system, the product boundaries, and its connections to adjacent systems. Non-functional requirements usually capture all types of other requirements like the visual properties of the system, its usability, and the performance requirements. Non-functional requirements may also include the product's intended operating environment and any maintainability, usability, risk analysis, portability and security issues. Non-Functional requirements also include cultural and political issues as well as legal requirements that the software must conform to [Bray, 2002]. Requirements are typically identified via use case scenarios, workflow descriptions and operational details provided by stakeholders [Van Lamsweerde, 2009]. Product requirement documentations that document the objective of the software and the capabilities and characteristics it should contain are the output of this phase and the input to the feature analysis phase. Going back to the definition of van Gorp, then identifying the requirements leads to identifying the features.

The second source of features is domain analysis, where features stem from analysing the requirements of a certain problem domain as defined by Gomaa [2005] “A feature is a requirement or characteristic that is provided by one or more members of the software product line”. Domain analysis can be distinguished from the single system approach in that its objective is to represent exploitable commonalities among many systems [Kang et al., 1990]. Domain analysis starts with identifying the scope of the domain. In order to create a feasible domain analysis, the boundaries of the domain should be clearly identified. This is done by identifying the problem scope or the problem domain. Next is the context analysis, which

defines the context of the domain, and the main concepts and characteristics of the domain are identified. This phase also includes identification of project constraints, and relations between the domain analysed and other domains. Sources of information for this stage include: domain experts, the study of similar domains, and the study of existing systems. In domain analysis, features represent concepts in the domain; the domain vocabulary should be used to name features.

Whether starting from analysing requirements of a single system that holds variation or performing a more thorough domain analysis for a family of systems (i.e. product line) or alternatively assimilating the features from both techniques, a feature represents a unit that will satisfy a certain requirement. Features can be found by mining for higher level *parts* that characterize the required software and identify its capabilities. These features will be *nouns* in the problem statement (e.g., documentation) that describes the required software characteristics, capabilities, or segments. For these high level characterizing features, it is important to keep a one to one mapping between requirements and features for the sake of obtaining well defined and consolidated features.

While identifying the features of the software, it is important to keep in mind that feature modelling is intended to identify the static capabilities of the software and indicate how these capabilities can vary; it is not intended for in-depth modelling of functionality and behaviour. Next to identifying the actual features, it is also important to identify how these features relate to the variability of the software. Gomaa [2005] states that in product description documents, statements using *could* often refer to *optional* requirements, while statements using *should* refer to *mandatory* requirements. We expand this statement into a more generic set of possibilities by which variable features can be identified. We do so by answering the following question: Where does variability come from? Several possibilities exist:

1. Variability may be noticed in advance and explicitly stated in the domain analysis phase. In this case, the different variants are identified by either defining different *alternatives* of a certain feature or defining a list of *could* include features.
2. Variability may come into the picture when the requirements leave some margin of freedom (coming from the “openness” of the requirements). In this case, many different possibilities exist to satisfy a specific requirement. In this case, the outcome is a set of possible features; these features differ in their importance and validity.
3. Variability may be used to represent prospective capabilities or business opportunities that are expected to be supported in the future, but are currently not yet supported either due to current hardware limitations, technology that is not yet supported or partially supported, time limitations, or cost limitations. In this case the corresponding features become optional features.

The variability analysis phase helps in identifying possible feature candidates and their contribution to variability. In addition, some information such as the involved stakeholders, links and dependencies between the features, and the granularity levels of the defined features should already be considered but not yet fully defined at this phase. Generally, the features anticipated in this phase will be of high granularity and represent the core functionality of the system. It is important at this phase to obtain a general understanding of the key features of the system and understand for which stakeholders they are important before proceeding to a more detailed modelling of the features identified. In the E-Shop product line, based on the requirements described and with the knowledge of the domain we could identify the following features: *Product Catalogue*, *Product Order*, *Shopping Basket*, *Payment*, *Electronic Delivery*, and *Shipping*.

The variability analysis phase provides the first brick in overcoming limitation L1, difficulties in using the feature modelling technique in practice. In the next section, we will

discuss how a more thorough analysis of the features, their composition, and their dependencies is established by analysing the features from a well-defined viewpoint using the multi-perspective approach.

6.4 Multi-Perspective Approach

Feature Assembly uses “*perspectives*” as an abstraction mechanism. We propose “*perspectives*” as an answer to our research question RQ1.3, *What kind of support can be provided during variability and commonality modelling to deal with large and complex systems?*

Often software can be considered from many different viewpoints (i.e. focus of interest), e.g., from the viewpoint of the user, from the viewpoint of the functionality, from the viewpoint of the hardware, etc. When modelling large and complex systems, trying to deal with all the viewpoints at the same point in time is very difficult, and will usually result in badly structured designs. Therefore, a more scalable approach is to identify the different points of view required to describe the software, and model the required capabilities (i.e. features) of the software with respect to one viewpoint (i.e. perspective) at the time. Usually, different stakeholders may use different perspectives (i.e. different points of interest). We define a perspective as:

Definition 6.2 (*Perspective*)

- ***A perspective is a particular focus of interest or context used for identifying the features composing the software.***

Not only do perspectives help in proving separation of concerns, they also provide an abstraction mechanism which allows focusing on only those features related to the perspective. This is important particularly in a complex domain and when many stakeholders are involved. These stakeholders may have a different focus or different modelling objectives. Separating the feature modelling process in terms of perspectives helps keeping the features of each point of view separate, therefore reducing the size of the models as well as the coupling between the models. Totally eliminating the coupling is not possible, as the features of the different perspectives are features of the same software and need to accomplish together the tasks and functionalities provided by the software. Therefore, the perspectives are linked to each other by linking their different features where needed. In Feature Assembly, the precise definition of the concept “*feature*” depends on the perspective it belongs to. We will provide definitions of the concept feature for different perspectives later on. Revisiting our definition of feature (definition 6.1), a feature within the multi perspective approach can be defined as:

Definition 6.3 (*Feature*)

- ***A feature is a physical or logical unit that acts as a building block for meeting the requirements or specifications of the perspective it belongs to.***

Within a single perspective, the Feature Assembly Modelling technique allows to formally represent how features are composed and related (section 6.5 presents the Feature Assembly modelling primitives). A feature belonging to one perspective may relate to other features in one or more other perspectives via *dependencies* (more on this in section 6.5.3).

Furthermore, the Feature Assembly modelling technique uses a *variable* and *extensible* set of perspectives. Each perspective describes the variability from a certain point of view (e.g.,

the User perspective, the Functional perspective, etc.), and together they should describe the variability of the required software. Therefore, the set of perspectives to be considered during the feature modelling process is *variable* (i.e. not predefined) and depends on the software to be modelled. This means that the modeller has to decide which perspectives are useful for describing the system and which not. The modeller can even stick to one single perspective (e.g., the System perspective) if all the variability falls in a single perspective. The Feature Assembly modelling technique is also *extensible* in the sense that if the modeller sees a certain perspective(s) that would help the modelling process, it can be added. The extensibility makes the technique flexible. While a set of perspectives would be adequate to model a specific product line, the same set may not be suitable for another product line.

To guide the modelling process, we have pre-defined a set of perspectives that could be used to model a software system. These include the *System* perspective, the *User* perspective, the *Functional* perspective, the *Non-functional* perspective, the *User Interface* perspective, a *Goal* perspective, and the *Persistent* perspective. As already mentioned, *it is not required to consider all these perspectives nor is the set of possible perspectives limited to only these ones*.

For instance, the *Persistent* perspective is only useful when modelling software that needs persistent data. As already mentioned, this set of perspectives is not fixed and can be further extended based on the needs of the application under consideration. For example, a *Hardware* perspective may be considered for embedded applications; a *Localization* perspective may be considered useful for software that needs to be localized for different markets; or a *Task* perspective may be considered useful for modelling task-based applications. The correct set of perspectives to use for modelling a system depends on how the requirements are identified. As already mentioned, one source of identifying features is by analysing the requirements.

Figure 6.2 shows the various perspectives that could be used to model the E-Shop application, namely the *System* perspective, the *User* perspective, the *Non-Functional* perspective, the *Functional* perspective, the *Graphical User Interface* (GUI) perspective, and the *Persistent* perspective. The lines linking the different perspectives refer to dependencies that exist between these different perspectives (in the form of feature dependencies as will be described in section 6.5.1). The solid lines refer to rigid dependencies, i.e. explicit dependencies that can be expressed by dependencies between features of the corresponding perspectives; while the dotted lines refer to soft dependencies, i.e. intrinsic dependencies that are anticipated from the understanding of how the system achieves its functionality but cannot be represented by dependencies between features. For example, a *Payment* feature in the System perspective will require a *Payment* feature in the Functional perspective indicating the different available options (i.e. possibilities) for payment. This will also influence the different payment options for an *Order Payment* feature in the Graphical User Interface perspective. The *Order Payment* Feature in the Graphical User Interface uses the *Payment* feature in the

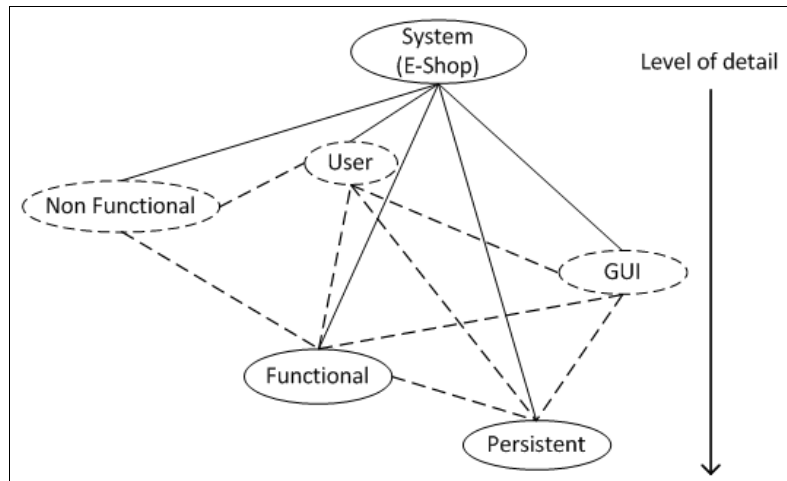


Figure 6.2: Overview of the different perspectives that could be used to model the E-Shop, and how they may relate to each other

functional perspective; this information is an example of an explicit dependency that should be explicitly represented in the Feature Assembly Models of this E-Shop. While, for example, an *Order Fulfilment* feature in the Functional perspective has no direct influence on the features of the Graphical User Interface perspective, but of course the type of order made and the type of available fulfilment for it will have some impact of the defined features of the Graphical User Interface.

It should be noted that the different perspectives may vary in the level of granularity used for the defined features, the more specific the focus of the perspective, the more fine grained the features will be. For example, in figure 6.2, the persistent perspective is very specific to features that have a persistence nature; therefore in general, the features will be of a higher level of detail than for instance features in the System perspective, which provides an overview of the features in the entire E-Shop application and therefore will be of the highest level of overview. As already explained, in each of these perspectives the definition of feature is different and based on the context of the perspective. For each of the defined perspectives, the features belonging to that perspective directly relate to the purpose of that perspective. For instance, features in the user interface perspective will contribute to the user interface of the system.

To provide some guidelines on how perspectives can be used to enhance the feature modelling process, in the next sections we will explain in more details how features could be identified for the following perspectives³¹: the *System* perspective, the *User* perspective, the *Functional* perspective, the *Graphical User Interface* perspective, the *Goal* perspective and the *Non-functional* perspective. For each of these perspectives we will explain the following: its purpose, when the perspective is used, how to find the features for the perspective, and provide an example. Furthermore, we will explain the *Persistent* perspective and how it can be used to create a variable data model in chapter 7.

6.4.1 System Perspective

- **Purpose**

The System perspective provides a high level overview of features that need to be delivered by the system/product line.

- **When Used**

This perspective provides a bird's eye view on the system. This overview is important to obtain an overall view of the size of the system (in terms of number of core features) and the amount of variability in the system's main features. The System perspective should already provide a good overview of the possible (product line) variants. It must be pointed out that a too detailed representation of the features would provide too much detail and thus the perspective would lose its objectiveness. On the other hand, a too shallow representation of the system's features would give an incomplete overview of the system which may lead to incomplete specifications of the system capabilities and how they can vary.

³¹ We will only highlight the most common perspectives, to illustrate the multi-perspective feature modelling concepts and ideas presented in this thesis

- **How to Find Features**

This perspective should only focus on high-level features that indicate the system's characteristics and capabilities. It is important to bear in mind that the intention is to model high-level features and pinpointing how they contribute to variability. The features in the System perspective adhere to the definition of feature as given by Kang et al. [1990]:

Definition 6.4 (*System Perspective Feature*)

- *A System perspective feature is a prominent and distinctive user-visible aspect, quality, or characteristic of the system.*

A feature in this perspective is a product characteristic that is self-contained, concise, distinctive, and user noticeable. In simple words, system perspective features answer the question *what capabilities does the system provide?* The System perspective's features are directly related to high-level requirements of the software.

Identifying high level features (i.e. key features) belonging to the System perspective is a twofold process; firstly, we mine the requirements (e.g., domain requirements, customer requirements, documentation, etc.) for candidate key features, i.e. fundamental features that make up the product. Secondly, once a key feature is identified, it should be defined how it contributes to the variability of the system (for high level features this should already be known from the variability analysis). Establishing a direct mapping between the underlying problem in the problem domain and the required system in the solution domain should result in high-level features being directly mapped from key requirements. As an example, a guideline for this mapping in the case of user driven application would be to look at the use cases³² or user scenarios³³; the subject of the use case identifies a key feature. We also note that not all use cases identify key features; this is because use cases may differ in their granularity. Rather the uses cases that *identify high-level system requirements* are those that identify key system features. In other words, uses cases that identify *scenarios* that are independent of other scenarios and have no prerequisite scenarios.

- **Example**

In the E-Shop product line, we might start by identifying the basic capabilities (e.g., based on the knowledge of the domain and previous domain analysis) of an E-Shop as the *Storefront* capabilities and the *Store Backend* capabilities; from there we might further drill down to elaborate more in order to explicitly identify the (core) features composing each of the above mentioned features of the E-Shop. As an example, the *Storefront* is composed of the following features: *Registration*, *Product Catalogue*, *Customer*, *Shopping Basket*, *Order Process*, and/or *Wish Lists*, *Customer Service*, and/or *User Tracking* capabilities. These are key features of the system that make up the overall systems functionality.

³² A use case is a list of steps, typically defining interactions between a role (known in UML as an "actor") and a system, to achieve a goal. The actor can be a human or an external system. [Kulak and Guiney, 2003]

³³ A scenario is a description of a person's interaction with a system. [<http://www.infodesign.com.au/ftp/Scenarios.pdf>]

Using the requirements as a source for identifying the features, the following use cases *Registration*, *Order Process*, and *Product Catalogue* map directly into key features, while the use cases *User Validation*, *Order Fulfilment*, *Product Display* are non-key features as they are actually respectively parts of the above-mentioned features. The next step would be to identify how these high-level features contribute to the variability of the required system; this is done following the guidelines mentioned in section 6.3. At this stage it is important to know whether or not the defined feature contributes to the variability of the system. As an example, in the E-Shop, *Registration*, *Order Process*, and *Product Catalogue* are all mandatory key features, while *Wish List* and *User Tracking* are optional features.

6.4.2 User Perspective

- **Purpose**

The *User* perspective represents the variability in the target users of the system. This information is relevant for understanding how the variability in users affects the variability in the features of the system.

- **When Used**

The User perspective is only relevant in systems/product lines that provide different functionality/products to different users or different groups of users. In this case the users affect the variability of the software through their influence on the selection and/or deselection of features. When the users of the system have an impact on the variability of the features (in one or more perspectives), this impact needs to be explicitly modelled. The Feature Assembly modelling approach represents these users (or actually user classes) as features. By doing so, the influence of the different user classes on the variability of the system can be made explicit. Therefore, in the corresponding Feature Assembly models, feature dependencies can be defined to represent how features are constrained to some specific users.

- **How to Find Features**

As already mentioned, in this perspective features represent different users using the system, and who have different needs and different concerns. Users (or actually the user classes) are mapped to features. Those features share the usage of the same set of system features, and influence their variability in the same way. The relations between them and the variations they hold are represented as well.

Definition 6.5 (*User Perspective Feature*)

- *A User perspective feature identifies a set of system users that share the same concerns and have the same influence on the variability of the system.*

- **Example**

A Quiz product line may have different features based on the target users. If used by business users it may be required to provide a self-assessment (i.e. used for employee training). While if used for schools there is a need for delivering both simple text based quizzes and more advanced quizzes and exams. Furthermore, the amount of

data stored is different for each type of user. For schools, there is a need for storing and reporting all information about users taking the quiz, while in the case of business usage this may not be necessary. In this case, the User perspective will hold the features *Schools* and *Businesses*.

6.4.3 Functional Perspective

- **Purpose**

The Functional perspective adopts a functional point of view when identifying features. It allows identifying features that represents the functionality provided by the system.

- **When Used**

Features belonging to the Functional perspective are features that have the role of defining what the system actually does. Functional perspective features can be considered as building blocks that help realize the system's functional requirements. It should be noted that the Functional perspectives identifies what capabilities are provided by the system, i.e. what the system does, but not how it does it. How the system performs its functionality should not be part of the feature model, and should be captured by other modelling techniques.

- **How to Find Features**

A good starting point for identifying the features belonging to the Functional perspective is the System perspective. High-level functionalities are already identified as part of the System perspective, the Functional perspective provides a more focused and detailed view on these features.

Furthermore, the Functional perspective helps understand the functional decomposition of the features belonging to the System perspective, taking into consideration the variability opportunities in this decomposition. On the one hand, it helps understand how the features contribute in order to provide the overall functionality of the system. On the other hand, it allows explicitly specifying possible variations (i.e. variation points and variants) of functionality.

Variability in the Functional perspective is related to the different methods for achieving certain functionality and the degree of variation to which the functionality should be realized; multiple possibilities could exist. A note to consider is that variability of Functional perspective features describes different possibilities available to achieve certain functionality; this is different than the runtime variability provided by the system, which refers to the dynamic response of the system as a respond to a certain action at runtime.

We define the concept feature in the Functional perspective as:

Definition 6.6 (*Functional Perspective Feature*)

- *A Functional perspective feature is a distinguishable well-defined functional characteristic of the system.*

Each feature should focus on only one functionality (action or verb) related to only one concept. If a feature represents a complex functionality (related to the same concept) then decomposing it should be considered. By a complex functionality we mean a functionality that can be further decomposed into finer grained functionalities. Decomposition is interesting only if it shows that this complex functionality holds a level of variation in how it can be satisfied. For example, in the E-Shop, one can define an Order Process feature that represents the activity of processing an order. This is a complex feature that could be split up into Order Transaction, Order Approval, Order Fulfilment and Shopping Basket, each of which represents a separate feature. A rather different case is when there are different possibilities to how certain functionality is performed. In this case, this functionality is mapped to a feature that represents a variation point. The different possibilities for how the functionality is performed are then considered as the variants of this variation point. For example, in the E-Shop application there could be three different varieties to how the shop products could be displayed, therefore one could identify a *Display Products* feature that represents this functionality of displaying products, and which is a variation point, that can be further decomposed into three variant features namely *Promotion Oriented*, *Product Oriented*, and *Customer Oriented* each with a different rationale for displaying the products. More details about decomposition and variants are given in section 6.5.1.

It must be noted that the Functional perspective may contain features common to other perspectives (e.g., the System perspective). This is actually an important characteristic of the Functional perspective as it represents the core functionality of the system and therefore it will contain features that also have presence in other perspectives. In particular the Graphical User Interface perspective (mentioned in the next section) and the Functional Perspective are very related, as in general the users will stir the functionalities of the system through the user interface. While the Graphical User Interface perspective focuses on the features from a user interface point of view, the Functional perspective focuses on the features from a “*service*” point of view. As an example, in the E-Shop application, in the Functional perspective, there will be a *sign in* feature to provide the functionality to sign in to the E-shop. There will also be a *sign in* feature in the Graphical User Interface perspective, but here the purpose is to show that the user should have a user interface component that allow him to sign in.

- **Example**

In the E-Shop, one can define an Order Fulfilment feature that is related to one thing, order fulfilling. Having known that there are three different possibilities of order fulfilment: service delivery, shipping, and electronic delivery, then the *Order Fulfilment* feature (which is then a variation point) will be split up into the features: *Service Delivery*, *Shipping*, and *Electronic Delivery* (which represent variants of this variation point). There are also two varieties for validating the shipping of an order, via the package slip or via the package tracking number. Therefore, the *Shipping* feature has two variants *Package Slip*, and *Package Tracking Number*.

6.4.4 Graphical User Interface Perspective

- **Purpose**

The Graphical User Interface perspective takes *visibility in the user interface* as a metric for identifying the features that should belong to it. In this perspective features are identified based on their availability in the user interface.

- **When Used**

This perspective should be used if variability is applicable to the user interface. Then, it becomes essential to explicitly state which parts of the interface are variable, and how the variable user interface elements relate to the variability in the system's functionalities, tasks, users, etc. Features belonging to the Graphical User Interface perspective are used to model the variability in the user interface. The Graphical user interface perspective lists the features that are subject to variation; whether they are interface elements (e.g. menus, toolbars, scrollbars, etc.) or features that have a visual presence (e.g. Sign Up, Shopping Basket, Favourites List, etc.). There may be various reasons to have variability in the user interface. Obvious reasons are due to variation in the application functionality, or variation in how different users view and interact with the application. Therefore, the variability in the user interface elements should be aligned with the variability of the features belonging to the other corresponding perspectives. This is achieved via using feature dependencies.

- **How to Find Features**

Features belonging to the Graphical User Interface perspective are visible for the user in the user interface and he/she can interact with them in order to activate some functionality of the system. Features within the Graphical User Interface perspective are defined as:

Definition 6.7 (*Graphical User Interface Perspective Feature*)

- ***A Graphical User Interface perspective feature is a visible and distinguishable user interface characteristic of the system.***

Features belonging to the Graphical User Interface perspective could be easily identified by analysing how the users will interact with the system, i.e. how they should initiate functionalities in order to accomplish their tasks, process information and respond to the system messages/functions. User interaction with the system is governed by the (core) requirements of the system; the (variable) functionality provided by the system impacts how users should interact with the system.

Therefore, features of the Graphical User Interface perspective can be found by analysing the use cases and/or user scenarios, which describe how the different users should interact with the system, and/or how different functionalities should be perceived by different users. In order to find user interface elements that are variable or are a source of variability, we can look to the type of the use case. Typically, each user would have a set of use cases that indicate how he/she uses the system. Each use case/scenario is in general associated with some information indicating the type of use case/scenario: a mandatory scenario (i.e. a scenario that will always happen), an optional scenario (i.e. it is one of other options that could happen), or an alternative

scenario (i.e. either this scenario will happen or some other scenario both can't happen together).

The Graphical User Interface perspective is used for identifying features perceived by the user in the user interface, and identifying how they are composed as well as the variability that is attached to them if any. Therefore only use cases with user interaction are the ones that are important for this perspective, how the system realizes these scenarios should not be considered in the Graphical User Interface perspective. On the other hand, realisation of these user scenarios is something to consider when looking for features in the Functional perspective.

As already mentioned, there is also a close connection between the features of the Graphical User Interface perspective and the Users perspective. This connection is made explicit in the Feature Assembly Modelling technique via feature dependencies and will be discussed in section 6.5.1.

It should be noted that the Graphical User Interface perspective is not intended to create a model of the Graphical User Interface; rather it captures features that characterize the units composing the user interface and their possible variations.

- **Example**

In the E-Shop we might imagine the following scenarios:

1. Users can navigate products by one of these techniques
 - By category
 - By searching
 - By similar products
2. Users can navigate the E-Shop in one of the following languages:
 - English
 - French
 - Dutch
 - Arabic
 - Chinese.

Looking at the first scenario, finding the features is quite straightforward. By searching for the nouns, we can identify the *Navigate Products* feature, the feature itself has no indication of variability therefore it is a mandatory feature. The *Navigate Products* feature has three different variations of how the products are visualized to the users, namely: *by category*, *by searching*, and *by similar products*. Therefore the Navigation feature is a variation point that has three variants namely: *Category*, *Search*, and *Similar Products*. The second scenario implies the need for localization of the E-Shop, i.e. localization of the text to the language of the interface and localization of the user interface components to suit the direction of the languages supported. Again searching for the nouns, we identify a *Localization* feature which is mandatory and has two parts, *Text Localization* which is the feature responsible for capturing the localization of the text and *UI Components Localization* which is the feature responsible for the localization of the user interface components (tables, lists, banners etc.) of the screen; both parts are mandatory features. The feature *Text Localization* can be further decomposed into two mandatory features, *Text Direction* and *Cursor Orientation*.

6.4.5 Goal Perspective

- **Purpose**

The Goal perspective allows defining features based on the understanding of the goals of the system. It is recommended for systems that use a goal-oriented requirements engineering method for identifying and electing the requirements.

- **When Used**

The Goal perspective is useful when a goal-oriented requirement analysis method is used to identify the system requirements. Opposed to functional analysis, goal-oriented requirement analysis focuses on early requirements phases, when alternatives (i.e. design decisions each of which can satisfy the initial goals) are being explored and evaluated.

A goal is defined by van Lamsweerde [2009] as “*a prescriptive statement of intent that the system should satisfy*”. During the requirements phase, it is considered important to understand the goals of the system because they answer the “*Why*” question of the system (i.e. “*Why do we need this system?*”). Goals give information related to the intension of the system and thus they implicitly or explicitly identify the various internal and external elements that affect the systems intensions. Goals that answer the *why* question are referred to as high-level goals, which define the intensions of the system. These intensions are further refined by asking “*How*” questions which might also trigger the “*How else*” questions which identify hidden or implicit variability. Goals that answer the *How* questions identify low-level goals.

- **How to Find Features**

In the Goal perspective features are found by analysing goals to gain an understanding of the influences that different requirements of the system have on one another, for example to identify conflicting goals (and therefore conflicting features). Goals are typically captured via goal models [van Lamsweerde, 2009] that analyses the goals of the system and how they relate to each other. Goals differ in their type as well as in their granularity; some goals identify business constraints and business requirements, while others identify the restrictions on the system operation. For example, in the E-Shop, the high-level goal “*allow online purchase of products*” is achieved via the sub-goals: “*allow product purchase*”, “*allow order submission*”, and “*allow customer tracking*”. Furthermore, goals help in pointing out both functional and non-functional requirements of the system. Goals identifying functional requirements are referred to as functional goals, goals identifying non-functional requirements (i.e. qualities) are referred to as soft goals. In the Goal perspective we define the concept of feature as:

Definition 6.8 (*Goal Perspective Feature*)

- ***A Goal perspective feature is a physical or logical unit that acts as a building block for satisfying a goal of the system.***

A Goal perspective feature takes the responsibility of satisfying an achievable atomic goal. An atomic goal is the simplest form of a goal. Non-atomic goals (i.e. those goals that can be further decomposed to simpler goals) map to composite features;

possible alternatives available by these goals map to possible variants that the feature may have. Another important issue related to variability is that through goals we can better define options for achieving these goals fully or partially. Therefore, it provides the opportunity to identify unforeseen variability in the domain (which may or may not be relevant due to cost limitations, time limitations, etc., but still remain important to identify for obtaining a complete domain description).

- **Example**

In the E-Shop application, we can distinguish the following high-level goal: *sell more products*, which can be achieved via any of the following three sub-goals: *promote new products*, *promote best-selling products*, and *promote products with high stock*. To define a mapping from the goals to features, an achievable goal is mapped directly to a feature, where the feature will take the responsibility of achieving this goal. In the example this will result into a *Product Promotion* feature, which has three different variants: *Best-Selling Promotion*, *New Products Promotion*, *High Stock Promotion*. In a similar manner, if we have the following goal for the E-Shop application: *feasible flexible shipping options*. This goal has the following sub-goals: *Flexible Shipping*, and *Meet Shipping Regulations*. This last goal is dependable on an external aspect of the system, namely shipping regulations, which may differ according to different locations. Furthermore, the *Flexible Shipping* goal is a high-level goal that is achieved via the following sub-goals *Quality Of Service Selection*, *Carrier Selection* and *Address Specification*. The *Flexible Shipping* goal is not achieved without the achievements of both its two sub-goals, therefore the goals will map to the following features: *Shipping Options* feature which is composed of the following features: the *Shipping Regulations* (which is an external feature that has impact on the behaviour of the system) and the *Quality Of Service*, *Carrier Selection* and *Address Specification* features.

6.4.6 Non-Functional Perspective

- **Purpose**

The Non-functional perspective adopts the viewpoint of defining all non-functional aspects of the system and models how they contribute to the variability of the system.

- **When Used**

A system may be influenced by non-functional aspects, also called quality aspects, which either facilitate or restrict how the system reacts to stimulations from the surrounding environment and/or from its users. Therefore, non-functional requirements can be a source of variability in which case it is important to explicitly represent them. Non-functional requirements often materialize soft goals and their impact on the variability of the system should not be neglected, due to the fact that they could influence a wide range of products.

- **How to Find Features**

Non-functional characteristics of the system could be an inherent part of the system, such as quality, security restrictions, or usability; alternatively they could be related to external parts of the system such as hardware limitations or recovery plans. Features that belong to the Non-functional perspective are defined to represent these characteristics. For this reason, we define the concept of feature in this perspective as:

Definition 6.9 (*Non-Functional Perspective Feature*)

- *A Non-Functional perspective feature is a non-functional characteristic of the system that contributes to or affects the system's variability.*

It is important to keep in mind to define only those features that contribute to the system's variability. For example, in the E-Shop application, a non-functional requirement *Response Time* with a maximum time value set to 5 milliseconds does not represent a non-functional feature as it contains no means of introduce variability. While, a non-functional requirement *Bandwidth Utilization* which defines the percentage of bandwidth utilized off the total bandwidth available, is mapped to a *Bandwidth Utilization* feature which has two variants, *Low* and *Medium*

It should be noted that features defined in the Non-Functional perspective will be related to features defined in other related perspectives such as the Functional perspective and the Graphical User Interface perspective. Such relations are maintained using feature dependencies as will be described in section 6.5.1.

- **Example**

In the E-Shop application, an important non-functional requirement is Security, which can vary according to the individual needs of each configured product. Security can be achieved via Credit Card Verification Codes (CCVC) and fraud detection services. Fraud detection services include: geographical IP address location checking (GIPLC), high risk IP address (HRIP), network post query analysis (NPQA), and e-mail checking. We map this information to a Security feature (which is a variation point) that has two variants *CCVC* feature and *Fraud Detection* feature. The Fraud detection feature has the following variant features, *GIPLC* feature, *NPQA* feature, *HRIP* feature, and *E-Mail* Checking feature.

6.4.7 Discussion

As already mentioned, in FAM we do not restrict the modeller to a particular set of perspectives. Rather, *we provide a set of possible perspectives as guidelines* for using perspectives when modelling variability. In the previous sections, we have provided details for the following set of perspectives: the *System* perspective, the *User* perspective, the *Functional* perspective the *Graphical User Interface* perspective, the *Goal* perspective and the *Non-functional* perspective. Furthermore, *this set can be further extended or scaled down*. A first reason to extend or scale down the perspectives is based on the needs of the domain of the application. For example, a Hardware Interface perspective is useful for embedded device product lines; a Persistent perspective is useful if the product line has some persistent features (i.e. features that contain information that should be stored).

Another criterion for selecting certain perspective and not selecting others is the viewpoint taken while defining the requirements of the system. Different approaches are

possible for this, e.g., a goal-based approach, use case driven approach, market driven approach, or user centric approach. As these requirements are used as a source to identify features, it is useful (and recommended) to use the same viewpoint during feature identification. Therefore, a goal perspective is recommended when a goal-oriented requirement engineering approach is/was used. This allows traceability between the requirements and the solution (i.e. the modelled product line). Therefore, the task-, functional-, and goal perspectives can be considered as alternatives. The system perspective provides the highest level of abstraction (or provides the least details); it is required for modelling any application and provides a good starting point for the modelling process, therefore we strongly recommend it as starting point. The other perspectives provide more details and represent a different way of looking at the system.

To help the modeller select the set of appropriate perspectives to use for modelling a certain product line, we have defined the *FAM Perspective Selection Process* illustrated in the flowchart shown in figure 6.3. The FAM Perspective Selection Process indicates that only one of the following perspectives should be used: Goal, Functional, or Task perspectives. For example, when using the Goal perspective there is no need to use the Non-Functional perspective, as the goal perspective captures both functional and non-functional characteristics of the system. While this is a recommendation to prevent modellers from using similar perspectives (which would result in a large amount of similarity between the identified features also possibly overlapped and redundant features), it is not a strict rule. Modellers may choose to arbitrary combine perspectives if it serves their needs (e.g., if a hybrid requirements engineering method is used to define requirements).

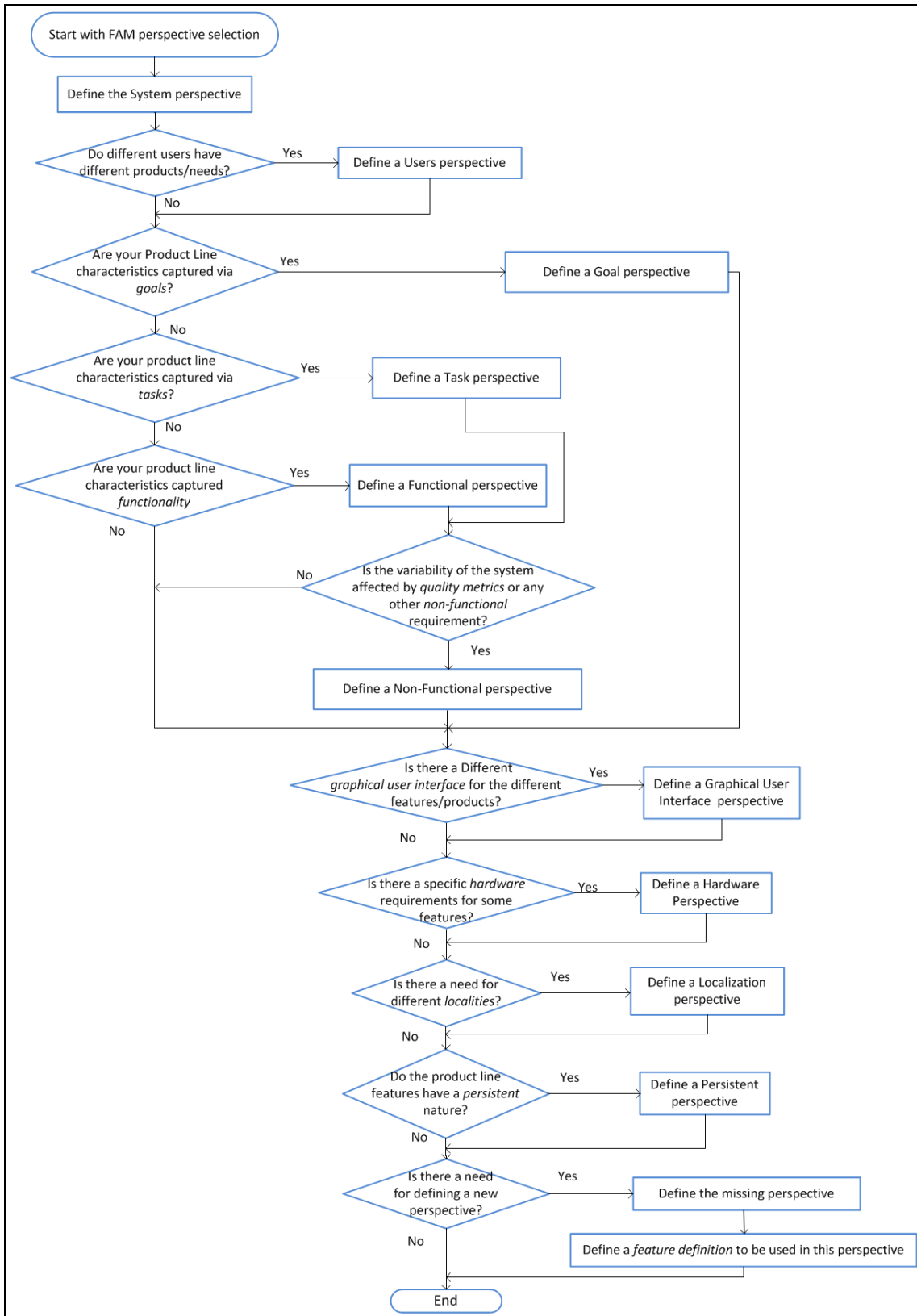


Figure 6.3: Feature Assembly Perspective Selection Process

6.5 Feature Assembly Modelling (FAM) Language

In this section, we present the Feature Assembly Modelling (FAM) language which allows to model features within a single perspective, resulting into a feature (assembly) model. The FAM language is the result of the study mentioned in chapter 5, it provides an answer to our research questions RQ1.2, RQ1.4 and RQ1.5 (see section 1.4) by overcoming the limitations mentioned in section 5.1 namely: limitations L1, difficulties in using the feature modelling technique in practice; L2, ambiguity in modelling concepts; and L4, limited reuse opportunities (section 5.1). For this purpose the FAM language was defined with limited modelling constructs and simple semantics in order to keep the modelling process as simple as possible (i.e. not doubts about which modelling construct to use). FAM uses a graphical notation to represent features, their relations, and their dependencies. Figure 6.4 shows the meta-model of the FAM language (using ORM notation). The basic construct is *Feature*; a *Product Line* (or variable software in general) is made up of a set of *Perspectives*. Each perspective represents a Feature Assembly Model and is of a certain perspective type (e.g., System, User, Functional, ...) (modelled as *Perspective Name*). A perspective type can only be used once for a certain Product Line. The feature assembly model of a perspective is made up of a set of features. We will discuss the modelling constructs in more details in the next sections. Note that an example elaborated with the FAM language will be given in chapter 8.

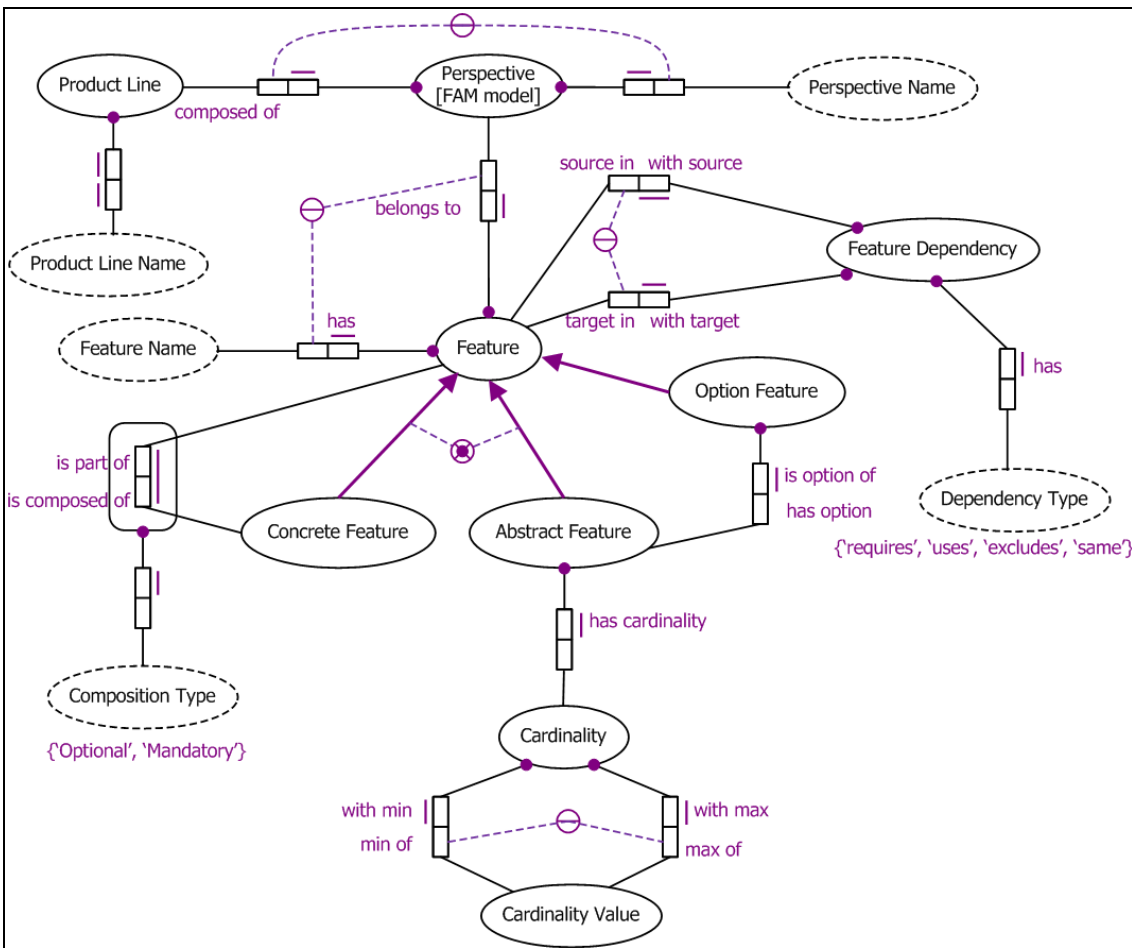


Figure 6.4: FAM Meta-Model

6.5.1 Features

In the FAM language two types of features are distinguished, *Feature* and *Abstract Feature*. Generally³⁴, a *feature* that represents a *concrete* and *well-defined* logical or physical unit or characteristic of the system will be referred to as “Concrete” Feature to distinguish it from the other type, being Abstract Feature. A concrete feature may be further decomposed into sub-features (in terms of concrete features or abstract features). A concrete feature is represented by a solid line rectangle holding the feature’s name. An *Abstract Feature* is a feature that is not concrete; rather it is a virtual feature that represents a *generalization* of some features or a *generalization* of certain characteristics or capabilities. An abstract feature will, in general, be associated with more specific features (concrete or abstract ones), of which it is a generalization. An abstract feature is used to indicate variability, it acts as a *variation point*, while the more specific features associated to it act as its *variants*. An abstract feature is represented by a dotted line rectangle holding the abstract feature’s name. Figure 6.5 shows the two feature notations.

To illustrate the difference between the two types of features, consider the E-Shop application. Features such as *Shopping Basket*, *Wish List*, *Order Process*, and *User Tracking* are all concrete features. The *Order Process* feature can be further decomposed into *Order Fulfilment*, *Order Approval* and *Order Transaction*. *Order Approval* and *Order Transaction* are both concrete features, while *Order Fulfilment* is an abstract feature. Indeed, order fulfilment is a generalization of two different types of order fulfilment (i.e. variants), namely: electronic delivery and product shipping. Therefore we define the *Order Fulfilment* feature is an abstract feature (i.e. a variation point) with two specializations (i.e. variants) *Electronic Delivery* and *Product Shipping*. Another example is found in a Quiz Product Line application. In this application, two types of operation modes are available: quiz operation mode and exam operation mode. Therefore, the *Operation Mode* feature is an abstract feature that can be further specified by two concrete features *Quiz* and *Exam*.

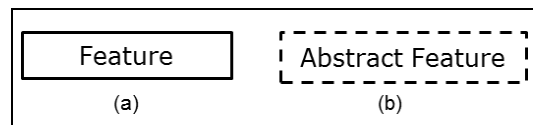


Figure 6.5: FAM feature notations: (a) “Concrete” Feature (b) Abstract Feature

It should be noted that the feature type as used in the FAM language (concrete vs. abstract) is different from the feature type used in mainstream feature modelling. In mainstream feature modelling, the feature type (i.e. mandatory, optional, AND, OR, and alternative) is used to express how a feature contributes to the variability. However, because a feature can contribute differently to variability in different situations, in FAM we do not associate such a variability type with the feature itself. Rather, how a feature contributes to the variability of the system is determined by the relations it has with other features. By doing so, reuse will be easier (more on this in chapter 9). Possible types of relations are explained in the next section.

A feature has a set of meta-data (i.e. properties) associated to it; these properties are not visualized in the model for the sake of readability but should be associated with the features³⁵. They are needed to complete the model and to facilitate information retrieval, model modification, etc. at later stages. These properties are:

³⁴ A more specific definition exists based on the perspective the feature belongs to as already mentioned in section 5.3.

³⁵ These properties are part of the Information model associated with the Feature Assembly Modeling technique represented by the FAM Ontology presented in chapter 10.

- **Name:** Each feature has a unique name within the perspective it belongs to. If two features belonging to the same perspective have the same name then this means they both represent the same feature, this criterion can be used to refer to the same feature in partitions of a model when it is too large.
- **Description:** the description is a descriptive explanation of one or two lines of the feature.
- **Owner:** The person that defined the feature; this is important to know in case of changes.
- **Stakeholders:** The persons that deal in some way with the feature (e.g., used it, have an interest in the feature, ...). For the sake of decision making it may be important to know for which stakeholders this feature matters. It may be necessary to distinguish between different types of stakeholders, however for the sake of simplicity we will not consider this here.
- **Keywords:** one or more keywords or tags may be associated to the feature as an index term or descriptor for later retrieval. The keywords associated to the feature should not be overloaded but should act as terms that captures the essence of the feature.
- **Binding Time:** The decision about the features included in the final product may occur at different stages in the development (e.g., design time, compile time, run time). Therefore, for a *variation point* (i.e. an abstract feature and a concrete feature with an optional composition relation) the binding time identifies the time at which this decision is taken.
- **Standalone:** this property indicates whether the feature is consolidated enough to be used “as it is”, i.e. independent of any other features. For example, features such as *Shopping Basket*, *Spelling Check* and *Equation Editor* could be reused independent of other features. It is important to strive for as much independency as possible because this will support feature reusability; standalone features are good candidates for being reused (more on this issue in chapter 9).

Furthermore, the name of the feature in combination with its perspective identifies the feature. The dot operator is used to define this *full identifier* of the feature. For example, a feature *Questions* belonging to the *System perspective* can be referred to as *System Perspective.Questions*.

6.5.2 Feature Relations

In FAM, we only use two types of feature relations: *composition relation* and *generalization/specification* relation. The distinction between these two feature relations is made to prevent ambiguities resulting from mixing feature compositions and variability compositions (please refer to section 5.1.2 for examples). We also restrict the possible types of relations depending on the type of the feature.

- **Feature Composition**

The composition relation is used to express a whole-part relation; i.e. a feature is composed of one or more fine-grained features. Composition relations are only supported for concrete features. The composition relation is either *mandatory* or *optional*.

A *mandatory* decomposition relation indicates a *compulsory* whole-part relation, i.e. the sub-feature must be part of all products derived from the model. An *optional* composition relation indicates an *elective* whole-part relation, i.e. the sub-feature may exist in some products derived from the model. A composition relation is graphically represented by a line with a diamond edge, the diamond points being at the composing feature (i.e. the whole). The line is a solid line for a mandatory composition relation (see figure 6.6(a)) and a dotted line for an optional composition relation (see figure 6.6(b)). Note that the *part* feature could either be a concrete feature or an abstract feature.

• Feature Specification

The generalization/specification relation is only allowed for abstract features. As already explained, an abstract feature is a generalization of some other features and the

generalization/specification relation is used to specify of which (specific) features the abstract feature is a generalization. In general, it is used to express a situation in which there is a need to distinguish a feature from the different possibilities or variants that it may have. The different *option features* (i.e. the specializations) of the abstract feature identify possible variants of that feature. In terms of variability, an abstract feature represents a variation point and the *option features* associated with it represent its variants. An option feature (variant) can either be a concrete feature or an abstract feature. The number of variant features allowed to be selected in a certain product is expressed via a *cardinality constraint*.

The cardinality constraint specifies the minimum and maximum number of features allowed to be selected in a valid product configuration, provided that the abstract feature is selected in the configuration. The notation used is “*minimum: maximum*”. A dash (“-”) is used to specify “any”, which means that there is no limitation on the maximum

number of variants that can be selected. The minimum cardinality should be greater or equal to one, while the maximum cardinality could be any integer between one and the maximum number of variant features. If only one variant should be selected then both the maximum cardinality and minimum cardinality should be set to 1 (equivalent to the alternative feature group in FODA). Moreover, a minimum cardinality set to one means that the abstract feature will be bound to one of its option features if it is selected in the product configuration. Note that the selection of the abstract feature itself is based on the type of feature relation it has with its parent feature (if any).

Figure 6.7 shows the FAM (visual) representation for the *Order Process* feature described in the previous section. The *Order Process* feature is a concrete feature that is

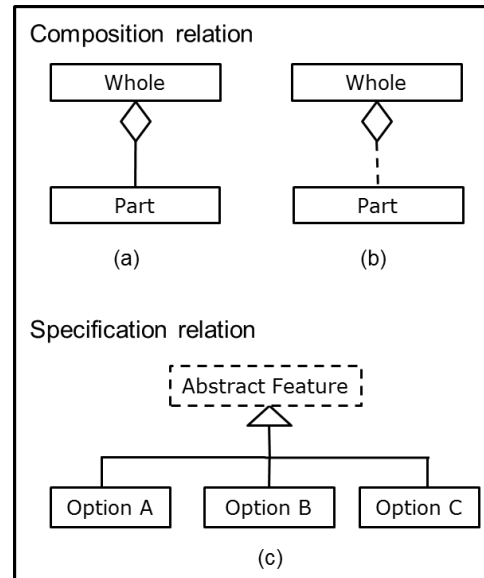


Figure 6.6: FAM feature notations (a) Mandatory Composition, (b) Optional Composition, (c) Generalization/Specification

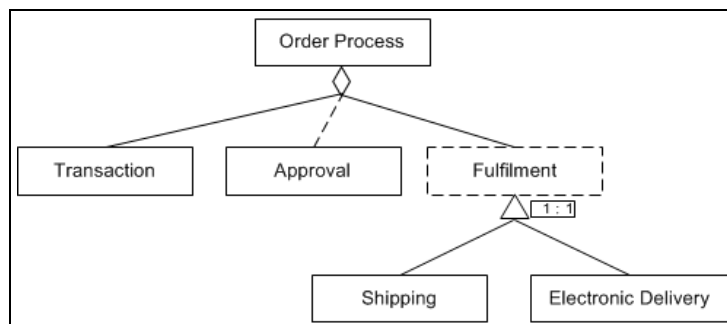


Figure 6.7: FAM representation of the E-Shop's Order Process feature

mandatory composed of a *Transaction* Feature and a *Fulfilment* feature; furthermore, it is optionally composed of an *Approval* feature. There are two types of fulfilment: shipping and electronic delivery; therefore the *Fulfilment* feature is an abstract feature that has two option features: *Shipping* feature and *Electronic Delivery* feature, both features are concrete. The *Fulfilment* feature is associated with a cardinality of minimum 1 and maximum 1.

6.5.3 Feature Dependencies

As already mentioned, feature dependencies capture and represent feature interactions, i.e. a feature dependency specifies how a feature may affect other feature(s). We argue (based on our study in section 5.2.1) that there is a need for expressive feature dependencies in which the reason for why the dependency holds is not lost. In the Feature Assembly modelling technique, *feature dependencies* are binary relations that allow expressing dependencies between features. We stick to binary relations because they are easier to grasp and understand than n-ary relations. On the other hand, n-ary relations may be more powerful. However they are more difficult to express by an average modeller, and could easily result into “non-elementary” n-aries, containing unnecessary information or redundant information. For example in the E-Shop application (figure 6.7), the dependency: `Electronic Delivery requires Approval AND (NOT Shipping)` is valid; yet since `Shipping` and `Electronic delivery` are alternatives, this dependency can actually be reduced to the following feature dependencies: `Electronic Delivery requires Approval`. Therefore, while n-ary feature dependencies might seem like a flexible nice-to-have utility, they increase the complexity of the resulted models and call for an additional step to “normalize” these feature dependencies. In order to keep our feature assembly models simple, we opt for the simplicity of the binary dependencies.

A feature dependency specifies how a feature may affect other feature(s). As already mentioned this could be due to a marketing requirement, business requirement, and domain constraint. Dependencies can be expressed between features from a single perspective as well as between features from different perspectives. As already mentioned in section 5.4, if there is a need to link together two perspectives, such a link is achieved by means of feature dependencies connecting features belonging to the different perspectives. We will explain below these two types of dependencies.





6.5.3.1 Feature dependencies within the same perspective

We have extended the set of feature dependencies defined in FODA³⁶ (*requires* and *exclude*) to more types of dependencies to better enable the modelling of feature interactions. Additionally, we found that although from a configuration point of view the need for specifying feature dependencies boils down to specify if, in a valid composition, a feature needs to be selected (e.g. requires dependency) or should not be selected (e.g. excludes dependency), from a modelling point of view the purpose of why the feature should be selected (or likewise deselected) is as important. Understanding why a feature requires or cannot be combined with another feature has great implications on the understanding of the overall systems and therefore will be useful for different kinds of decisions. Furthermore, this information may also be essential in the case of change or future evolution for the system (e.g. if is based on domain constraints, technical difficulties, or marketing preferences). Therefore we associate each feature dependency with a *Reason* which holds a textual description for the purpose of the

³⁶ For more details please refer to chapter 2.

dependency. Additionally, like for features, it may be interesting to know who identified this dependency, therefore we add an *Owner* property to each feature dependency.

A feature dependency takes the form: *Feature_A* <feature dependency type> *Feature_B*. Dependencies among features of the same perspective are called *inter-perspective dependencies*. Listing 6.1 shows the graphical representation and the associated semantics of the feature dependencies supported by FAM. The feature dependencies can be specified in the feature model graphically by connecting the relevant features with a line containing the graphical notation of the dependency (examples are shown in chapter 8) or using a text form.

 Excludes	<p><i>Feature A excludes Feature B</i> indicates that Feature A and Feature B cannot occur together (are mutual exclusive) in a valid product. As an example, <i>Maximum Graphics</i> excludes <i>Maximum Performance</i>; <i>Single Licence</i> excludes <i>Multiple Choice Questions</i>.</p>
 Requires	<p><i>Feature A requires Feature B</i> indicates that Feature A is dependent on Feature B, and likewise Feature B is dependent on Feature A; so <i>A requires B is the same as B requires A</i>. In terms of configuration, the Requires dependency implies that if feature A is selected in a valid configuration then feature B must also be selected and the other way around. As an example, <i>Advanced Editor</i> requires <i>Spelling Checker</i>.</p>
 Uses	<p><i>Feature A uses Feature B</i> indicates that feature B is required for feature A to achieve its service or capability; i.e. Feature A needs Feature B for some of its functionality. This is an asymmetric property (thus the arrow is the symbol), so Feature A uses Feature B does not imply Feature B uses Feature A. While in a valid configuration the selection of Feature A triggers the selection of Feature B, the existence of Feature B does not imply the existence of Feature A. As an example <i>Search</i> uses <i>Display Products</i>.</p>
 Same	<p><i>Feature A same Feature B</i> indicates that the two features are the same. This dependency is particularly important in the case of very large feature assembly models of which parts were developed independently; the <i>same</i> dependency allows <i>gluing</i> them. The "same" dependency can be considered as a merge operator that enables merging perspectives based on their common features.</p>

Listing 6.1: FAM Feature Dependencies, notations and semantics.

Following the semantics given in listing 6.1, we see that some of the feature dependencies are *symmetric*: *excludes*, *same*, and *requires*. This implies that the direction of the dependency is not significant, and therefore no direction is associated with their graphical representation. The *uses* feature dependency is *asymmetric* thus a direction (i.e. arrow) is associated with the graphical representation of it. Furthermore, the feature dependencies: *requires*, and *uses* are transitive properties, i.e. if Feature A *requires* Feature B, and Feature B *requires* Features C, then Feature A *requires* Feature C.

6.5.3.2 Feature dependencies between different perspectives

As already explained, in FAM a perspective oriented abstraction mechanism is used while modelling, yet perspectives are not independent. Features belonging to different perspectives may be dependent. It is often the case that a feature in one perspective constrains

another feature belonging to a different perspective. Dependencies among features of different perspectives are called *intra-perspective dependencies*. For intra-perspective dependencies we support the same set of dependencies as for inter-perspective dependencies (*excludes*, *same*, *requires*, *uses*). The form of an intra-perspective dependency is: $\langle \text{Perspective.feature} \rangle \langle \text{feature dependency type} \rangle \langle \text{Perspective.feature} \rangle$, where $\langle \text{feature dependency type} \rangle$ is one of the keywords: *excludes*, *same*, *requires*, *uses*. Here a feature must be identified by both the name of its perspective and its feature name. An example intra-perspective dependency representing interdependencies in the E-Shop application is: *Functional.Promotion requires User_Interface.Discount*, which states that supporting the *Promotion* (functional) feature requires having the *Discount* feature in the user interface. Similarly *User_Interface.Discount uses Functional.Discount_Rate*, states a uses dependency between the user-interface perspective and the functional perspective. Intra-perspective dependencies are also associated with a *Reason* and an *Owner* properties so that the rationale of the dependency is not lost.

6.5.4 FAM Formal Specification

In this section we present a formal specification of the Feature Assembly Modelling Language. According to [Harel and Rumpe, 2004] a modelling language consists of an abstract

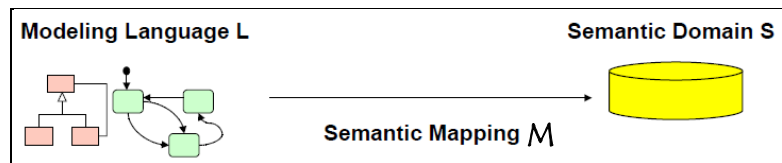


Figure 6.8: Semantic Definition of a Modelling Language, after [Grönniger et al., 2009]

syntax (L), which defines the allowed constructs by the language (the symbols and their formulation rules); and semantics (S) which describe the meaning of the language constructs, such that $M: L \rightarrow S$ (as represented by figure 6.9). We build on top of this and define the syntactic language L_{FAM} which represents the Feature Assembly Modelling Language, and provide the semantics by means of the mapping M_{FAM} ; i.e. $M_{FAM}: L_{FAM} \rightarrow S_{FAM}$. We describe each in more details below.

6.5.4.1 FAM Syntax

As already shown in the previous sections, FAM is a visual modelling language (i.e. has a visual notation). Feature Assembly models are therefore diagrams which represent graphs. Following the definition of Erwig [1998] for a visual language, a visual language L_{FAM} over an alphabet A consists of a set of symbols of A that are, in general, related by several relationships $\{r1, \dots, rn\} = R$. Thus we can say that a diagram d is given by a pair (s, r) where $s \subseteq A$ is the set of allowed symbols of the diagram, and $r \subseteq s \times R \times s$ gives the relationships that hold in d . In other words, d is nothing but a directed graph with edge labels drawn from R , and a visual language is simply a set of such graphs. Language semantics definitions are often based on so-called *abstract syntax* which defines a language on a more abstract level and can safely ignore all aspects that are not needed within the semantics definition [Erwig 1998]. That is why we can abstract from the choice of icons or symbols and from geometric details such as size and position of objects (language symbolic notation are given in sections 6.5.1, 6.5.2 and 6.5.3). Therefore, in this section we only restrict ourselves to the abstract syntax defining the Feature Assembly modelling constructs, i.e. the language L_{FAM} .

We first start by defining the set of allowed symbols s in the diagram, and the set of allowed relations r .

Definition 1: Language Symbols

The alphabet of valid symbols of the language consists of:

- (1) A non-empty finite set of feature symbols, i.e. nodes of the diagram.

$$\mathbf{Features} = \{f_1, f_2 \dots f_n\}.$$

The set *Features* is partitioned into two disjoint sets *ConcreteFeatures* and *AbstractFeatures*:

$$\mathbf{Features} = \mathbf{ConcreteFeatures} \cup \mathbf{AbstractFeatures}$$

- (2) For each feature $f_i \in \mathbf{Features}$, there is a constant symbol \mathbf{FName}_i , $\mathbf{FName}_i \in \mathbf{FNames}$, $\mathbf{FNames} \subset \mathbf{String}$; i.e. for each set of features $\{f_1, f_2, \dots, f_n\}$ there exists a corresponding set of feature names $\{\mathbf{FName}_1, \mathbf{FName}_2, \dots, \mathbf{FName}_n\}$. To refer to the name of a feature $f \in \mathbf{Features}$, we use the notation $\mathbf{FName}(f)$.

- (3) A non-empty finite set of perspective names *PNames*.

$$\mathbf{PNames} = \{\mathbf{PName}_1, \mathbf{PName}_2, \dots, \mathbf{PName}_n\}, \mathbf{PNames} \subset \mathbf{String}.$$

- (4) A non-empty finite set of product line names *PLNames*.

$$\mathbf{PLNames} = \{\mathbf{PLName}_1, \mathbf{PLName}_2, \dots, \mathbf{PLName}_n\}, \mathbf{PLNames} \subset \mathbf{String}.$$

- (5) The set *Cardinality* which is a set of ordered pairs, such that

$$\mathbf{Cardinality} = \{(\mathbf{Cmin}_1, \mathbf{Cmax}_1), \dots, (\mathbf{Cmin}_n, \mathbf{Cmax}_n)\}, \quad \text{where}$$

$$\mathbf{Cmin} \in \mathbb{N}, \mathbf{Cmax} \in (\mathbb{N} \cup \{-\}), \mathbf{Cmax} \geq \mathbf{Cmin} \text{ iff } \mathbf{Cmax} \in \mathbb{N}$$

The cardinality pairs will be used to identify the minimum and maximum cardinality allowed for an abstract feature.

□

The language relations refer to the set of possible relationship between the features; these include feature relations (composition relation and specification relation) and feature dependencies. This is defined as follows.

Definition 2: Language Relations

- (1) A *Composition* relation is a binary relation that links concrete features to their sub-features. There are two types of compositions: mandatory composition, and optional composition, therefore *Composition* is partitioned into two *disjoint* sets *MandatoryComposition* and *OptionalComposition*.

$$\mathbf{Composition} = \mathbf{MandatoryComposition} \cup \mathbf{OptionalComposition}$$

where,

$$\mathbf{Composition} \subseteq \mathbf{ConcreteFeatures} \times \mathbf{Features};$$

$$\mathbf{Composition} = \{(Cf, f) | Cf \in \mathbf{ConcreteFeatures}, f \in \mathbf{Features}\}$$

- (2) A **Specification** relation is a ternary relation that links abstract features to their list of (option) features and associating this with the cardinality of this specification.

$$\mathbf{Specification} \subseteq \mathbf{AbstractFeatures} \times 2^{\mathbf{Features}} \times \mathbf{Cardinality};$$

$$\mathbf{Specification} = \{(Af, \{f_1, f_2, \dots, f_m\}, C) \mid Af \in \mathbf{AbstractFeatures}, \{f_1, f_2, \dots, f_m\} \subseteq \mathbf{Features}, Af \notin \{f_1, f_2, \dots, f_m\}, C = (Cmin, Cmax), C \in \mathbf{Cardinality}, \text{ where } Cmax \leq m \text{ if } Cmax \in \mathbb{N};$$

- (3) A **Dependency** relation is ternary relation that expresses a dependency between two features. Each dependency has a type and is a directed edge, i.e. an edge between a source feature (**sf**) and a target feature (**tf**), where **sf** \neq **tf**.

$$\mathbf{Dependency} \subseteq \mathbf{Features} \times \mathbf{Features} \times \mathbf{DType};$$

$$\mathbf{Dependency} = \{(sf, tf, dtype) \mid sf \in \mathbf{Features}, tf \in \mathbf{Features}, dtype \in \mathbf{DType}, tf \neq sf\}$$

DType is the finite set that represents the valid dependency types. It is partitioned to two disjoint sets: **ADtype**, which represents the asymmetric dependency types and **SDtype** which represents the symmetric dependency types

$$\mathbf{DType} = \mathbf{ADtype} \cup \mathbf{SDtype}; \text{ where } \mathbf{ADtype} = \{ \mathbf{uses} \} \text{ and } \mathbf{SDtype} = \{ \mathbf{excludes}, \mathbf{requires}, \mathbf{same} \}$$

We further require that for each tuple (**sf**, **tf**), there is at most one dependency **dtype**.

□

Next, we define the set of syntactic rules (i.e. formulation rules) that cover the construction of well-formed feature assembly models and perspectives, as follows.

Definition 3: Feature Assembly Model

A Feature Assembly Model **fam** is a tuple:

$$\mathbf{fam} = (\mathbf{F}, \mathbf{R})$$

where,

$$(1) \mathbf{F} \subseteq \mathbf{Features}, \mathbf{F} \neq \emptyset$$

$$(2) \mathbf{R} = (\mathbf{C}, \mathbf{S}, \mathbf{D})$$

Where $\mathbf{C} \neq \emptyset$ or $\mathbf{S} \neq \emptyset$

$$\mathbf{C} \subseteq \mathbf{Composition} \text{ and } \mathbf{C} \subseteq \mathbf{F} \times \mathbf{F},$$

$$\mathbf{S} \subseteq \mathbf{Specification} \text{ and } \mathbf{S} \subseteq \mathbf{F} \times 2^{\mathbf{F}} \times \mathbf{Cardinality},$$

$$\mathbf{D} \subseteq \mathbf{Dependency} \text{ and } \mathbf{D} \subseteq \mathbf{F} \times \mathbf{F} \times \mathbf{DType}$$

To refer to the features of a Feature Assembly Model fam , we use the notation $F(fam)$; to refer to the composition relations, specification relations, and dependency relations of the Feature Assembly Model fam , we use the notations $C(fam)$, $S(fam)$, $D(fam)$ respectively.

□

A perspective corresponds with a feature assembly model; i.e. it is defined by a feature assembly model. Therefore a perspective is defined as a Feature Assembly Model together with the name of the perspective.

Definition 4: Perspective

A perspective *Perspective* is a tuple:

$$\mathbf{Perspective} = (fam, PName)$$

where,

- (1) fam is a Feature Assembly model
- (2) $PName \in PNames$

To refer to the name of a perspective, we use the notation $PName(Perspective)$ and to refer to the Feature Assembly Model of the perspective, we use the notation $fam(Perspective)$.

□

A product line consists of a set of feature assembly models, one for each perspective, together with the name of the perspective.

Definition 5: Product Line

A product line *ProductLine* is a tuple:

$$\mathbf{ProductLine} = (PLName, \{Perspective_1, \dots, Perspective_n\})$$

where,

- (1) $PLName \in PLNames$
- (2) $Perspective_i$ is a perspective.
- (3) $\forall Perspective_i \in \{Perspective_1, \dots, Perspective_n\}, \forall Perspective_j \in \{Perspective_1, \dots, Perspective_n\}, i \neq j: PName(Perspective_i) \neq PName(Perspective_j)$
- (4) $\forall Perspective_i \in \{Perspective_1, \dots, Perspective_n\}, \forall f_i, f_j \in fam(Perspective_i), i \neq j: FName(f_i) \neq FName(f_j)$

The set $\{Perspective_1, \dots, Perspective_n\}$ contains all perspectives used for product line *ProductLine*.

□

6.5.4.2 FAM Formal Semantics

The semantics (i.e. interpretation) of our language constructs is given in terms of configurations.

The ultimate purpose of a Feature Assembly model is to understand the available variable features that a product line could hold, in order to guide how products can be composed (i.e. configured) from the defined product line. Therefore, it actually represents a constrain problem of which its possible solutions represent valid products.

We will first define the configuration for a Feature Assembly Model, next the configuration of a Product Line.

Definition 6: Configuration of a Feature Assembly Model

A configuration for a feature assembly model $fam = (F, R)$ where $R = (C, S, D)$, $Conf$, is a subset of features, $Conf \subseteq F$, on which the following rules hold:

- (1) $Conf \neq \emptyset$
- (2) *if $f_i \in Conf$ and $\exists f_j \in F: (f_j, f_i) \in C$
then $f_j \in Conf$*
- (3) *if $f_i \in Conf$ and $\exists f_j \in F, f' = \{f_1, \dots, f_n\} \subseteq F, cr \in Cardinality:$
 $i \in \{1..n\}, (f_j, f', cr) \in S$
then $f_j \in Conf$*
- (4) *if $f_i \in Conf$ and $\exists f_j \in F: (f_i, f_j) \in C$ and $(f_i, f_j) \in$
MandatoryComposition
then $f_j \in Conf$*
- (5) *if $f_i \in Conf$ and $\exists f' = \{f_1, f_2, \dots, f_n\} \subseteq F; cr = (a, b) \in$
Cardinality: $(f_i, f', cr) \in S$
then $\exists f'': f'' \subseteq f'$ and $a \leq |f''|$ and $(|f''| \leq b \text{ if } b \in \mathbb{N})$ and $f'' \subseteq Conf$*
- (6) *if $f_i \in Conf$ and $\exists f_j \in F: (f_i, f_j, requires) \in D$ or $(f_j, f_i, requires) \in$
D
then $f_j \in Conf$*

(7) if $f_i \in \mathbf{Conf}$ and $\exists f_j \in F: (f_i, f_j, \mathbf{excludes}) \in D$ or $(f_j, f_i, \mathbf{excludes}) \in D$
then $f_j \notin \mathbf{Conf}$

(8) if $f_i \in \mathbf{Conf}$ and $\exists f_j \in F: (f_i, f_j, \mathbf{same}) \in D$ or $(f_j, f_i, \mathbf{same}) \in D$
then $f_j \in \mathbf{Conf}$

(9) if $f_i \in \mathbf{Conf}$ and $\exists f_j \in F: (f_i, f_j, \mathbf{uses}) \in D$
then $f_j \in \mathbf{Conf}$

□

A configuration for a product line may be derived from the configurations of the Feature Assembly Models of each perspective defined for the software product line. The configuration can be defined as the union of the configurations of the different feature assembly models fam .

Definition 7: Configuration of a Product Line

Let $PL = (pname, \{(fam_1, pname_1), \dots, (fam_n, pname_n)\})$ be a product line.

Let $Conf_1 \dots Conf_n$ be configurations of the feature assembly models $\{fam_1, \dots, fam_n\}$, then a configuration $Conf$ for PL is defined as follows:

$$Conf = Conf_1 \cup Conf_2 \cup \dots \cup Conf_n$$

□

Variability Notations

Next we provide the syntax and semantics of the *variability* notations, i.e. *variation points* and *variants* as follows:

(1) **VariationPoints** represents a non-empty finite set of variation points, i.e. nodes that denote variability in the diagram,

$$\mathbf{VariationPoints} = \{VP_1, VP_2 \dots VP_n\}.$$

(2) **Variants** represents a non-empty finite set of variations, i.e. nodes that denote variations in the diagram,

$$\mathbf{Variants} = \{V_1, V_2 \dots V_n\}.$$

Definition 8: Variation Points and Variants

(1) $f \in \mathbf{AbstractFeatures}, (f, f', c) \in \mathbf{Specification}, f' = \{f_1, f_2, \dots, f_n\}, |f'| \geq 1, c \in \mathbf{Cardinality}$

then $f \in \mathbf{VariationPoints}$, and $\forall f_i \in f' : f_i \in \mathbf{Variants}$, i.e. $f' \subseteq \mathbf{Variants}$

(2) if $f_1 \in \mathbf{ConcreteFeatures}$, $f_2 \in \mathbf{Features}$,
 $(f_1, f_2) \in \mathbf{OptionalComposition}$

then $f_1 \in \mathbf{VariationPoints}$

□

6.6 Discussion

In this chapter, we explained the FAM approach. Compared to the mainstream feature modelling techniques (mentioned in chapter 5), we have introduced an abstraction mechanism based on perspectives to deal with the cognitive difficulty of modelling large and complex systems, and we tried to overcome the limitations of the mainstream feature modelling techniques by limiting the number of modelling concepts and by having a rigorous separation between composition and variability. We discuss these two contributions in more detail by comparing them to existing solutions.

1. Perspectives as Abstraction Mechanism

As already mentioned, separation of concerns improves the design of complex and large systems. One of the concerns for Feature Assembly Modelling was to support variability and commonality modelling of large and complex systems (RQ1.1, and RQ1.3). FAM uses a perspective-based approach to separate concerns and allow in this way to focus on one aspect at the time. Furthermore, the intra-perspective dependencies allow linking the different perspectives. In addition, the modeller may opt for an arbitrary number of perspectives. This is opposed to the technique of categorizing features adopted in FODA (which groups features using predefined categories - see section 5.1.4 for more information). First of all, it is not always easy to decide on the category of a feature, and secondly, it is not an abstraction mechanism but rather a grouping mechanism with a fixed number of groups, which may make it hard to decide to which group a feature belongs. Figure 6.9 and figure 6.10 illustrate the difference between the two approaches using the Private Branch Exchange (PBX) system [Kang et al., 2002].

Using FODA (shown in figure 6.9), one model is created to represent the overall system. Such a model can be very difficult to create when the number of features is large and may become difficult to understand. A predefined fixed set of categories (also called layers) is used to capture the different types of features of the system. Features are grouped together by means of the following predefined set of categories: capabilities, operating environments, domain technology, and implementation techniques. Features are related using the “implemented by” dependency. A feature belongs to only one category, as shown in figure 6.9.

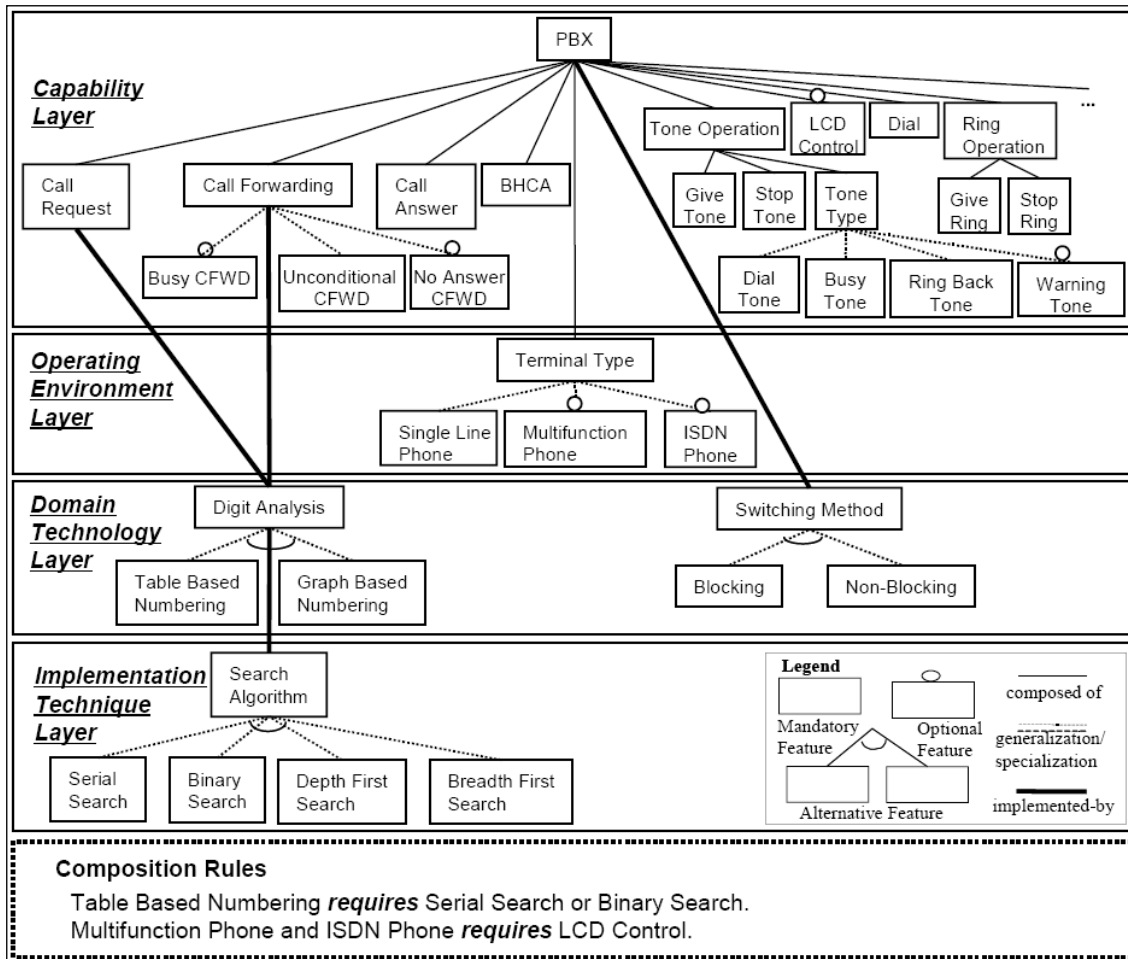


Figure 6.9 FODA model of PBX problem

Using FAM (shown in figure 6.10), different models are used to model the system from the viewpoints of the different capabilities of the system. Here we opted for a *System Perspective*, a *Hardware Interface Perspective*, a *Functional Perspective*, and a *Non-Functional Perspective*. For each perspective, separate feature models are created to model the variability and commonality of features that represent the capabilities of the application from that specific point of view (note that features common between two or more perspectives are shaded).

While we have four perspectives in FAM, the feature models in these perspectives are small and as such easier to understand, and easier to create as one only has to focus on one aspect of the system at a time. All feature in a perspective also server the same purpose; this comes from the fact that in FAM “what is a feature?” is answered based on the purpose of the perspective the feature is part of. Therefore using perspectives as an abstraction mechanism also reduces the difficulties of using the feature modelling technique in practice by providing clear guidelines for what could be a feature and what not (see section 5.1, limitation L1.1, L1.2)

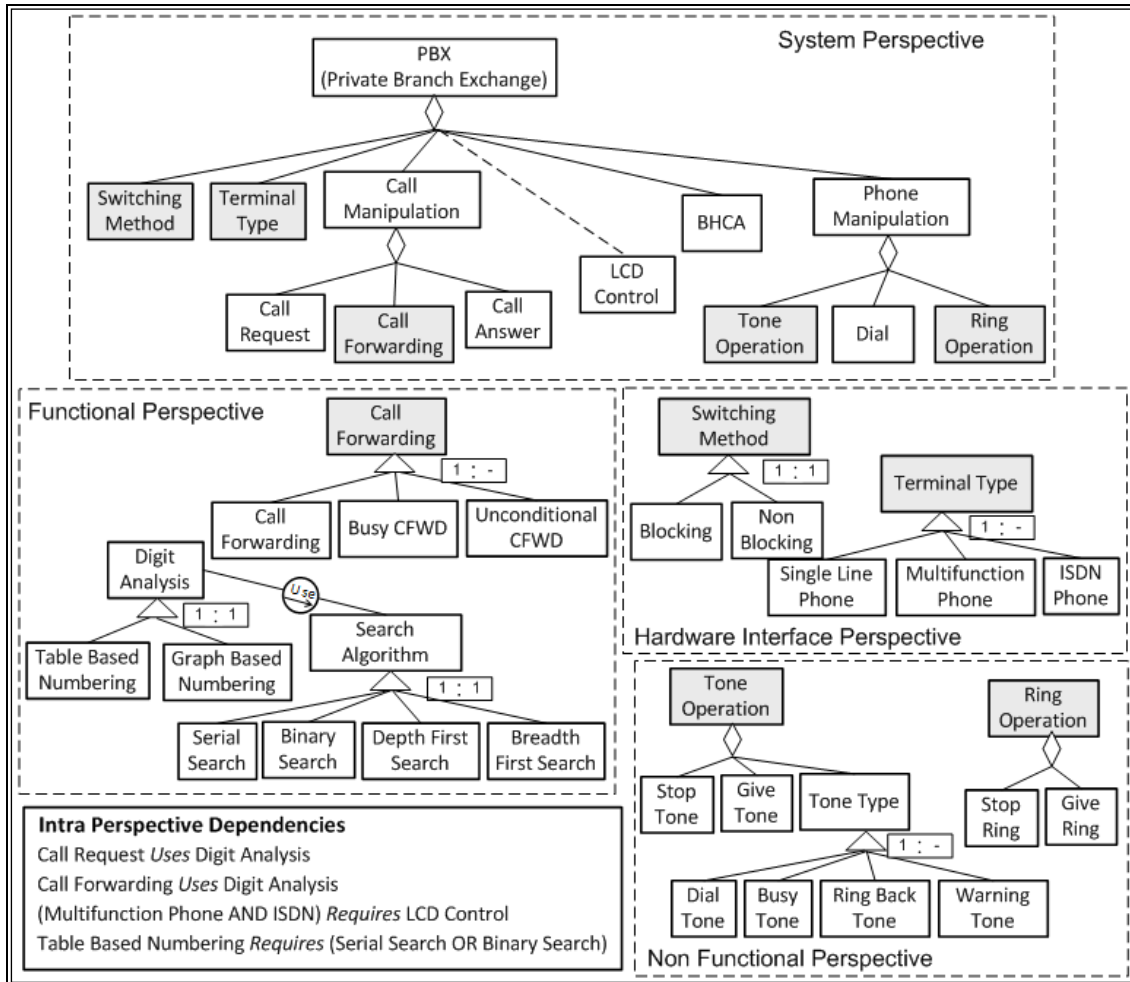


Figure 6.10 FAM model of PBX problem

2. Well-defined Modelling Semantics

Another concern for FAM was to alleviate the feature Modelling practice (this is related to our research questions RQ1.2, RQ1.4, RQ1.3), we addressed this concern by dealing with the limitations of mainstream feature modelling techniques mentioned in section 5.1 (limitation L2 and L4). The FAM language uses modelling concepts and notations that reduce the creation of ambiguous models (Limitation L2, please refer to section 5.1.2 for more details). By introducing *Abstract Features* to capture variability (variation points), practitioners are *forced to make all information explicit* in their models. Figure 6.11 demonstrates this; figure 6.11.a provides the FODA representation of the Graph Product Line problem (GPL) (introduced by [Lopez-Herrejon et. Batory, 2001]), figure 6.11.b provides the FAM Representation of GPL. In Figure 6.11.a, the feature Graph Type is mandatory, but it is not clear whether one has to select an alternative feature from only one of the alternative feature groups (Directed-Undirected) and (Weighted, Unweighted) or one has to select an alternative feature from each of these groups. This ambiguity comes from the fact that in mainstream feature modelling techniques there is no distinction between decomposition and specification relations. Thus the *Graph Type* feature (in figure 6.11.a) is decomposed into two sets of specification branches (feature groups). This representation is not possible in FAM. In FAM there is a distinction between features that can be decomposed and features that can be further specified, combining the two is not possible. Therefore to model the GPL problem presented in

figure 6.11.a in FAM, there is a need to explicitly introduce new features that represent the information that was implicit in the feature model of figure 6.11.a. This resolves the previously mentioned ambiguity of the GPL in figure 6.11.a. In the FAM representation of the GPL, shown in figure 6.11.b, the *Graph Type* feature is characterized (via a specification relation) into *Direction* and *Weight* features. This specification is associated with the cardinality of minimum 1 and a maximum of any. The *Direction* feature has two specifications (*Directed-Undirected*), associated with the cardinality of minimum 1 and maximum 1 (i.e. equivalent to an alternative relationship). The *Weight* feature has two specifications (*Weighted, Unweighted*) associated with the cardinality of minimum 1 and maximum 1.

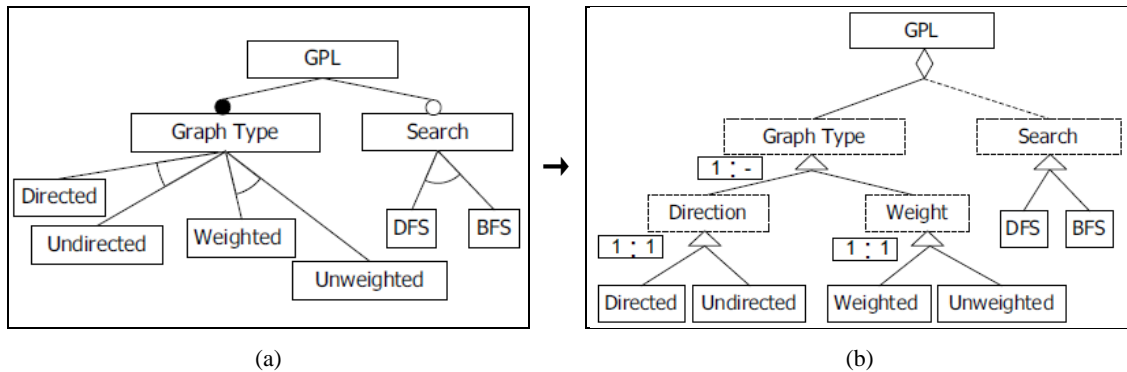


Figure 6.11 (a) FODA Representation of the GPL (b) FAM Representation of GPL

In addition, by explicitly distinguishing between composition relations and specialization relations, the modelling constructs cannot be overloaded, and therefore *we avoid the need for normalization* and avoiding the redundancy caused by allowing both singleton relations (optional and mandatory) and group relations (e.g. AND, OR, and alternative) (please refer to section 5.1.2).

Another advantage of distinguishing between concrete features (which can only participate in composition relations) and abstract features (which can only participate in specialization relations) is that it frees a feature from the information of how it contributes to variability allowing to easily reuse the feature with different variability requirements thus supporting feature (and feature model) reuse. Figure 6.12 demonstrates this; figure 6.12.a shows the FODA feature model for a *Payment* feature, which has two alternative features *Bank Transfer* and *PayPal*, the equivalent in FAM is shown in figure 6.12.c, *Payment* is represented as an abstract feature which has two option features *Bank Transfer* and *PayPal*, associated with a cardinality of maximum and minimum of 1 (i.e. equivalent to the alternative variability of FODA). In successive product lines the *Payment* feature needs to be extended with other payment methods, e.g., *Visa*, *Mastercard*, and *Bancontact/Mister Cash*, as shown in figure 6.12.b. Furthermore, suppose that the *Bank Transfer* feature needs to become mandatory to suit all markets while there is a need to select one or more of the other payment features (OR Features). Such a change requires deleting the old Alternative Feature group, creating a new OR group, and changing the type of the *Bank Transfer* feature to mandatory, the result³⁷ is shown in figure 6.12.b. In FAM this is a simpler process as shown in figure 6.12.d, the new

³⁷ Note that adding and removing branches in the feature model tree may not always be a straightforward task in current tools (e.g., it may need backtracking and reconstruction of more than one branch or even level)

payment methods are added as new options for the Payment feature, the cardinality is changed to a minimum of 1 and a maximum of any to suit the new situation. The need for the *Bank Transfer* feature to become mandatory is actually a constraint rather than an intrinsic fact of the domain, therefore this is modelled using the feature dependency relation “Payment requires Bank Transfer”, as shown in figure 6.12.d. This example shows that FAM supports managing change in existing feature models as well as supporting reuse of features (more on FAM’s support for reuse is given in chapter 9)

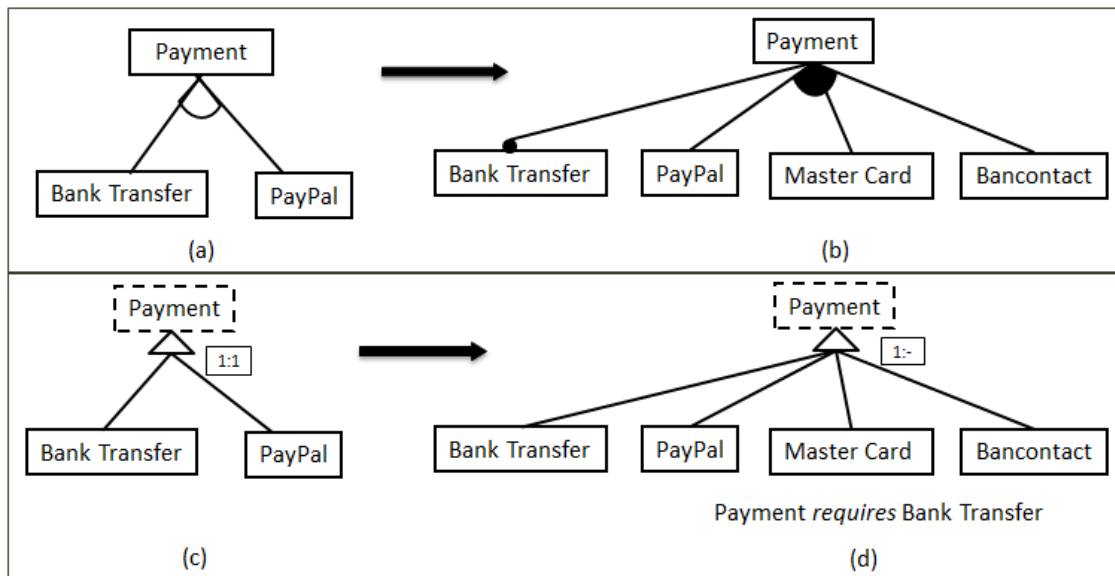


Figure 6.12 Support for changes and feature reuse a comparison between FODA and Feature Assembly Modelling

6.7 Summary

In this chapter, we presented one of the main contributions of this thesis, the Feature Assembly Modelling (FAM) technique. To guide the modellers, we have presented some guidelines for analysing variability and identifying variable features, i.e. features that indicate variability. Next, we have presented the multi-perspective approach and the FAM language that allows creating feature assembly models within the different perspectives specified.

We have supported modelling with abstraction by using a multi-perspective approach for feature modelling. Perspectives act as abstraction mechanism enabling separation of concerns when modelling. Adopting a perspective-based approach for defining features allows identifying the features that are relevant for a particular aspect or viewpoint, thus acting as an abstraction mechanism that helps dealing with complexity. Furthermore, dealing with one concern at a time allows for better scalability in the case of very large systems. We have also presented guidelines for how features can be identified in the different perspectives. We provided these guidelines for the most common perspectives; the same principles apply for any perspective that may be useful for modelling the system. Furthermore, by expressing dependencies between features of the different perspectives, the different perspectives are interconnected, which provide a more complete picture of the system modelled.

The FAM language overcomes some of the limitations of the mainstream feature modelling techniques. In FAM, we have reduced the number of modelling primitives used and

more importantly the specification of the information about the variability is separated from the definition of the features, which improves reusability. We have also provided a link between variability concepts (i.e. variation points and variants) and the modelling constructs so that anticipating variability from the created feature models is a straightforward process. We show how features can be related to each other in terms of feature-to-feature dependencies. We defined a set of feature-to-feature dependencies that can be used for features of the same perspective as well as for features of different perspectives (i.e. intra-perspective dependencies). Intra-perspective dependencies allow putting all the perspectives together in order to obtain the complete picture of the system. We concluded the chapter with providing examples to demonstrate how FAM has overcome the limitations of the mainstream feature modelling techniques that were mentioned in the previous chapter.

Chapter 7

Feature Assembly Modelling For Data Intensive Applications

Data intensive applications are applications that manipulate a large amount of persistent³⁸ data. In general, the data intensive applications provide an interface by which users can manipulate the underlying data, but it is also possible that the data is only for internal use by the application. For developing efficient data intensive applications an alignment between the application's functionality and the data on which it operates is required. In some cases, more than one application share access to common data entities, yet these different applications do not necessarily share the same view on these data entities. Therefore, in data intensive applications, an adjustment between the data and the application(s) responsible for manipulating this data is required. For efficiency purposes (e.g., if the data is shipped with the application, or the database is distributed) as well as for security (the application should only access the portion of the data that it is authorized to operate on), it may be required that the application only has access to the data entities that it actually needs, this is usually achieved via materialized views and/or virtual views [Garcia-Molina et al, 2008]. Design decisions about which entities should be part of which view, should be made to optimize the performance and reusability of the data intensive applications.

In the case of introducing variability to data intensive software applications, the alignment between the application (in this case a member of the product line) and the data is even more crucial. Members of the product line vary in the features (i.e. capabilities and characteristics) that each product provides and different features may be associated with different parts of the persistent data processed, generated, or accessed by the application. So, different combinations of features may trigger different combinations of data entities. Therefore, in the case of variable software, it should be possible to provide variability at the data level in order to have a correspondence between the variable application and the database schema. Different features selected for the final product may imply different views or even a different database scheme. Therefore, for each possible product it should be possible to tailor an adequate database view (physical or virtual) that provides only the data entities that are needed for the features used within a specific product. That is, as the product line is configured to provide a set of valid products the *data schema* should also be configurable. For example in an E-shop application, a shop that does not support a *Wish List* should not have the *Wish List* data entity as part of its database.

³⁸ Generally speaking, the persistent data could be held in a relational database, an object oriented database, an XML file, or in a lightweight tailored DBMS (e.g., in the case of embedded devices).

To realize this, we define the concept of a *variable data schema*. A variable data schema is a data schema that holds optional data entities that may or may not be partially or fully included in a product's final data schema. A variable data schema also contains variability annotations that annotate database entities with variability information (see section 6.3).

In a certain product derived from a product line, existence or absence of features accordingly motivates the selection or absence of related persistent data entities. Yet the process is not straightforward, as it implies injecting knowledge about the applications variability into the database design stage. Moreover, it also implies providing traceability links to document how features relate to database entities. To be able to achieve this, variability modelling needs to be extended to support persistent variable data modelling. Therefore, there is a need to support the following tasks: 1) the link between persistent data entities and application features should be expressed explicitly; 2) the database entities should be designed in such a way that these variable schema entities can be selected/deselected in a flexible manner.

To achieve this, the Feature Assembly Modelling technique has introduced the *persistent perspective* which allows identifying features that have a persistent nature and allows modelling their variability. The persistent perspective is an intermediate link between the database model and the application's (variable) features. In the coming sections we explain how features are modelled in the persistent perspective. Next, we explain how to maintain the link between Feature Assembly models and the data models, in order to take into consideration variability information in the process of database design.

7.1 The Persistent Perspective

To allow modelling variable data intensive applications (or product lines) using the Feature Assembly Modelling technique a *Persistent perspective* is defined. The Persistent perspective holds features that have a persistent presence in the application. These persistent features capture the persistent data in the other perspectives of the application. Features in the Persistent perspective are linked to features in other perspectives (e.g., System perspective, Functional perspective, etc.).

It must be noted that the Persistent perspective is not a conceptual model of the data used by the application. Rather it is a variability model of features representing data concepts (and thus the data entities that exist within the application). However, it is incapable of representing a complete conceptual schema of the underlying database because it lacks the notion of *relation*, which is essential to express how the different data concepts are related. Being a conceptual *variability modelling* technique, Feature Assembly models are not suitable for data modelling. Therefore, the variability knowledge contained in the Feature Assembly model should still be linked to the conceptual models created during data modelling³⁹.

For example, in the E-Shop application, a *Purchase Order* feature is defined in the System perspective because it represents a key feature in such an application. Additionally, a *Purchase Order* feature should also be represented in the *Persistent perspective* because the information about a purchase order should be persistent⁴⁰. This also implies that the corresponding data model should hold a *Purchase Order* entity that represents the purchases

³⁹ This could be done using a data modelling technique such as EER or ORM, in this thesis we illustrate the use of EER to create a variable schema.

⁴⁰ Persistent could refer to any type of persistency, e.g., file or database. In this chapter, we will use persistent to refer to database persistency.

made by a certain user at a certain moment (timestamp). Like in the other perspectives, features in the persistent perspective should be organized in a Feature Assembly Model(s) to represent the variability of the different persistent features. As such, it also indicates the required variability in the underlying resulting schema.

7.1.1 Defining the Persistent Perspective

The Persistent perspective holds features that denote *persistent data*. Such a feature represents a feature (or concept) within the application that has a persistence nature, e.g., is stored in the database. A feature in the persistent perspective is defined as:

Definition 7.1 (*Persistent Perspective Feature*)

- ***A Persistent perspective feature is a feature that has a persistent presence in the application.***

It should be noted that the level of detail in the Feature Assembly Models of the Persistent perspective depends on the level of variability in the corresponding product line. For example, in a product line that targets many users with different roles and at different localities, the same information may be named differently based on the context (i.e. function and user role) and on the locality (i.e. the same terms may have different names). This variability should be indicated in the features defined within the Persistent perspective. The Persistent perspective represents features that directly manipulate concepts stored in the database. It is important to note that features defined in the Persistent perspective should contribute to the variability of the overall application.

The features of the Persistent perspective are motivated by the persistence nature of application features, and therefore they stem from features within the different other perspectives of the applications (e.g., System perspective, Functional perspective, Users perspective, Non-functional perspective). The features of the Persistent perspective are based on all the other features that exist in the other perspectives defined for the product line. Therefore, the Persistent perspective should be defined after the features of the product line have been identified and analysed for completeness. In addition, features of the Persistent perspective will hold a strong relation with features defined in these perspectives, i.e. expressed by means of feature dependencies. We define the following methodology for identifying and defining features that should belong to the Persistent perspective.

1. **Identify in the System perspective the features that represent or require persistent information.** For these features it will be necessary to introduce corresponding features into the Persistent perspective. For example in the E-Shop application, in the System perspective some features are directly concerned with persistent information such as the features *Product*, and *Product Category*. While others indicate the need for persistent features because they manipulate persistent data, for example the *Shipping Order* feature indicates the shipping information about a certain *Purchase Order* of a certain *user*. From this we can derive the need for a supporting feature in the Persistent perspective, i.e. a *User* feature. Additionally, it also indicates the need for a Persistent *Purchase Order* feature.
2. **Define composition relations between the persistent features** to define how they relate to each other from a compositional point of view. Also define the nature of these compositions, i.e. mandatory or optional. As already mentioned, a mandatory composition means that the sub-feature is an indivisible part of the parent feature; while an optional composition means that the sub-feature is complementary to the parent feature.

3. **Investigate the need to introduce or distinguish between abstract and concrete features.** A persistent feature may be a generalized concept that has several more specific types; in this case this feature is defined as an (persistent) abstract feature. The more specific variants of that concept are represented as (persistent) option features; they are linked to the more general concept in terms of a generalization/specification relation. Next, define the cardinality rules that govern the maximum and minimum number of option features that should be selected in a valid product configuration.
4. **Define the feature dependencies** that relate the features in the Persistent perspective to features in the System perspective
5. **Repeat steps 1 to 3 for all perspectives** (Functional perspective, Users perspective, etc.) to extract and define all persistent features. In each iteration, go through steps 2 and 4 to extend the feature model defined so far with additional features derived from the other perspectives, and with additional feature dependencies.

7.1.2 Refine the Persistent Perspective

As mentioned before, the main reason for defining a Persistent perspective is to enable a better understanding of the variability of the system/product line and how that affects the persistent data associated with the different product line instances. The features defined in the Persistent perspective in addition to the feature-to-feature dependencies (between features of the Persistent perspective and features of other perspectives) should be taken into account when defining variability in the database schema. Furthermore the Persistent perspective should help tie together the functionality of the system with the persistent data manipulated and stored. Due to the often-tangled relation between data and functionality, a refinement for the features defined in the Persistent perspective is required, taking into account the features and dependencies defined in the Feature Assembly models created in all perspectives. This refinement is a two-step process:

1. Validate the consistency of the Persistent feature assembly model and the associated inter-perspective dependencies and refine when necessary. This means verifying the following:
 - a. Check if no persistent features are missing. This can be done by going through the persistent concepts that need to be defined within the product line.
 - b. Check whether some features have the same semantics and actually represent the same feature (this can occur because the features may originate from different perspectives). In case there is a need for this duplication the “same” dependency should be used to indicate that they are the same features. For example, in the E-Shop product line, two features with the name *Shopping Basket* may exist, one derived from the Functional perspective and the other derived from the Graphical User Interface perspective. In each perspective, the *shopping basket* has a persistent nature; therefore a persistent *Shopping Basket* feature is defined for each. But actually, they both refer to the same information and there is no need to duplicate the information and therefore only one persistent feature *Shopping Basket* should be kept as part of the persistent perspective.
 - c. Complete inter -dependencies between features.
 - d. Check whether no conflicts exist in the dependencies defined (more on this in chapter 10).

2. Validate the feature-to-feature dependencies within the different perspectives (both inter-dependencies and intra-dependencies) and refine when necessary. This means verifying the following:
 - a. Complete the intra-dependencies between features of the different perspectives and the Persistent perspective. For example, in the E-Shop product line it is important to relate both the *Shopping Basket* feature from the Graphical User Interface perspective and from the Functional perspective to the *Shopping Basket* feature of the Persistent perspective via the following dependencies: *Functional.Shopping Basket* same *Persistent.Shopping Basket*; *Graphical User Interface.Shopping Basket* uses *Persistent.Shopping Basket*
 - b. Check whether no conflicts exist in the intra-dependencies defined.

A good Persistent perspective should contain all necessary persistent features that features of all other perspectives need to manipulate. Once the Feature Assembly models are defined, the next step is to use this information during the database modelling process in order to obtain a variable database schema which is compatible with the different possible products defined in the product line.

7.2 Linking Feature Assembly Models and Data Models

A conceptual data model is a database design that is independent of the implementation of the actual database (i.e. RDBMS, performance issues, security issues, etc.). The Persistent perspective provides a link between the features of the product line along with their variability opportunities, and the (required or existing) underlying conceptual data schema. Table 7.1 shows the analogy between Feature Assembly Modelling concepts and conceptual data modelling concepts. This analogy helps in defining a mapping between *features* and *entities* (in EER modelling). It should be noted that variability could not only affect entities but also attributes. Furthermore, two scenarios exist when defining the link between the data model and the variability model. Firstly, it is possible that the data model is small and thus a centralized data model can be used [Connolly and Begg, 2009]. In a centralized data design the data model is defined in one design step, and as a result one global database model is defined. Different views on the data can then be defined if required. Alternatively, it is possible that the data model is large and multiple users are involved, in that case a decentralized database schema is used based on the different user views [Connolly and Begg, 2009]. In the decentralized data design, a data model is defined for each user view. In case a global data model is required, it can be defined via a view integration process where the different segments of data design are combined to create one global model. In either case the Feature Assembly models of the Persistent perspective can be used to guide the data modelling to produce a variable data model that is compatible with the different variability needs of the product line. We will discuss each approach into more details in sections 7.2.1 and 7.2.2.

	Feature Assembly Model	Data Model (EER)
Concepts	Feature: a physical or logical unit that acts as a building block for meeting the specifications of the perspective it belongs to	Entity: is any distinguishable object or concept that is to be represented in the database. It is the representation of a 'thing' in the real world.
		Attribute: represents a property or some characteristic of the entity it belongs to.

Assemblies	Composition: A feature can be composed of a set of sub-features (mandatory composition or optional composition)	Aggregation: an entity is characterized by a set of attributes that represent properties of this entity.
Generalization	Abstract Feature: An abstract feature that denotes a type or category of features (its sub-features).	Generalized Entity: An entity that combines general characteristics of a group of entities.
Specification	Option feature: defined by a <i>variant</i> relationship.	Subtype defined by a <i>is-a</i> relationship.
Relations	Dependencies: a dependency between two or more features.	Relation: a relationship among two or more entities (represents an associative property; integrity constraints are examples of possible relations.)

Table 7.1: Relation between feature assembly model concepts and data modelling concepts

To specify the variability aspects, we have extended the EER model with annotations that are used to mark the variability of the schema entities, attributes, and relations. To denote variability of an entity or attribute, it is annotated with the annotation `<<variable>>`, i.e. it is dependent on the selection of a corresponding feature in the product line. To denote variability due to generalization/specialization entities, we use the annotation `<<variant>>`. The annotation `<<variant>>` indicates entities that were derived from an option feature. A variable data concept should be further annotated with information about its source of variability, i.e. the (variable) features to which it is related. Two annotations are used to denote this link between the features in the Feature Assembly models and data concepts in the data model, namely `<<maps_to>>` and `<<relates_to>>`. This allows traceability between the features and their corresponding data concepts. Table 7.2 explains these annotations and their semantics. It should be noted that data concepts could be related to any feature within any perspective.

Annotation	Semantics
<code><<maps_to>></code>	Identifies a <i>one to one mapping</i> between a feature assembly model concept (i.e. feature) and a data model concept (i.e. entity or property). E.g., <code>Persistent.Questions <<maps_to>> Data_Model.Questions</code> <code>Persistent.Passing_Score <<maps_to>> Data_Model.Quiz.Passing_Score</code>
<code><<relates_to>></code>	Identifies a descriptive <i>association relation</i> between a Feature Assembly model concept (i.e. feature) and a data model concept (i.e. entity or attribute). E.g., <code>Functional.Question Category <<relates_to>> Data_Model.Category</code> <code>Functional.Add Question Assessment <<relates_to>> Data_Model.Assessment</code>

Table 7.2: Annotations denoting relations between features and database concepts

7.2.1 Linking Features to Data Entities - The Centralized Data Model Approach

In case of applications where the database users all share the same view on the database, a centralized data model may be the best option. In this case, the only views required on the data are the views derived for the different products of the product line. These views can

be physical views (i.e. the tables and properties are actually extracted from the global data schema to meet the requirements of a certain product) or virtual views (i.e. views are saved in the data schema).

To allow the design of a variable data schema (i.e. a schema that can be easily tailored to meet the variability of the application), the following rules can be used during data modelling:

1. Features in the Persistent perspective map to data entities in the data model.
 - a. Key features, i.e. features that represent a concrete concept or object, map to entities. For example: *Persistent.Products* <<maps_to>> a *Products* entity in the data model.
 - b. Features expressing details of key features are mapped to attributes rather than entities. For example *Persistent.Price* <<maps_to>> the *Price* attribute of the *Product* entity.
2. Persistent variation points, due to an optional compositional relation (part-of), trigger variability in the underlying database schema. Those variant features are mapped to the appropriate data concept (entity or attribute) and variability is indicated with the <<variable>> annotation. For example, a *Product* feature can be optionally composed of *Product Details* (e.g., image, weight, dimensions, colour, size, brand, etc.) in this case the *Product* entity is linked to a variant composite entity named *Product Details* which holds as attributes the details for a certain product. In this case, *Product Details* is marked with the <<variable>> annotation to indicate this variability (it semantically means that not each E-Shop has details associated with its products).
3. Persistent variant features (i.e. children of an abstract feature) should be mapped to separate persistent concepts in the data model (i.e. entities). Each variant feature is mapped to a separate entity. This ensures having a flexible schema in which not all the (variant) entities need to be selected. The variable entities or attributes should be marked with the <<variant>> annotation. For example in the E-shop Persistent perspective, there could be four different variants of *Product*: *Consumer Products*, *Application products*, *Services products*, and *E-Book products*. Therefore, these four variants features are mapped to four entities (each feature is represented by an entity); each annotated with the <<variant>> annotation to indicate their variability.

A good variable schema design should take into account the need for separating <<variable>> and <<variant>> concepts in order to have a schema that could be easily customized for each possible product. For strong <<variable>> entities, this is easily achieved by either selecting or deselecting the whole entity. For example, in the E-Shop product line, if a certain product configuration will be using the E-Shop to sell Books and E-Books, then this means that the corresponding schema will only have the entities that relate to the *E-Book* and *Consumer Products* (in this case Book products). Weak <<variable>> entities often provide additional information or more details about a certain strong entity. If a weak entity is selected, its corresponding strong entity should also be selected. Variable attributes could also be easily deselected if not used as an index or primary key. For example, in this E-Shop product, the entity *Product Details* will also be used, selecting only the attributes that are suitable for books, namely (image, weight, dimensions). Therefore for good variable schema design, <<variable>> and <<variant>> properties should not be used as primary keys or indexes. A foreign key relationship will also not be valid in the case that the <<variable>> or <<variant>> attribute is not selected.

7.2.2 Linking Features to Data Entities - The Decentralized Data Model Approach

In large systems, database modelling becomes a lengthy process in which there are multiple needs from different users (or applications) interacting with the database. The requirements of each should be identified and often a different user view is provided for each database user (or group of users). This allows providing customized views to each user, shielding him/her from the complete data model. View modelling is defined by Navathe and Schkolnick [1979] as “the modelling of the usage and information structure of the real world from the point of view of different users and/or applications”. It must be noted that these different database users (or applications) are well identified during the data design and have well known and fixed requirements. In large systems, it is actually the case that view design is established as a first step towards obtaining a global⁴¹ data design. The design starts with analysing each view from the point of view of its (database) users. View integration methodologies are then applied to these segmented views in order to have a global schema [Batini et al., 1986].

In the case of variable software, the different users of the database, which are in this case the products that could be configured from the product line, are not known beforehand. In variable software the variables are bound later in the design, deployment or even at runtime, leaving open all possible valid configurations. Therefore, it should be possible to customize a data schema at any time in the development process (production time, installation time, or runtime, depending on when the binding of variables happens). Note that in this case there are two different sets of *views* affecting the data model. Firstly the *user views* (i.e. the result of the view modelling process) that is, different users see different portions of the data. Secondly, the *variability views* (i.e. the result of the variability modelling process) that is, different variability requirements imply different data requirements. The different user views will be reflected in the variability view via defining the Users perspective. The Users perspective allows indicating the availability of the different features for the users. In essence, variability may occur in each user view; therefore it should be possible to combine the variability views with the user views for each specific user (user-groups). This combination allows us to directly use the different perspectives as a source for detecting persistent concepts that belong to each specific user-group. User views are reflected in each perspective by the set of dependencies that are used to relate the features within the perspective with the user group information that is modelled in the Users perspective.

For each user view, variability information can be added to the entity relationship models defined (similar to process described in the previous section). View integration techniques could be used to obtain a global conceptual schema as explained in [Batini et al., 1986].

In Chapter 8, the approach explained to use Feature Assembly modelling to define variability in persistent data, will be demonstrated with an example. We also demonstrate the process of identifying a variable schema.

⁴¹ In distributed databases, a global data design is not required

7.3 Summary

In this chapter, we have presented an approach for modelling data variability. The introduction of variability in software development also has its impact on data intensive applications. The option “one data schema fits all” is usually not a desired solution. The variability introduced in data intensive applications should not only operate on the functional side, but variability should also be introduced in the database schema. We call a data schema where variability is introduced a *variable data schema*. A variable data schema is a schema that can be easily tailored to meet the requirements of the different variants of the product line. It should be noted that variability in the application triggers variability in the database. Therefore it is important that there is a link between both.

The Feature Assembly Modelling technique can be used to model variability in persistent data. A *Persistent perspective* is defined in which persistent features are defined along with their relations to the features in the other perspectives. The persistent features can be identified by inspecting the features in the other perspectives used for modelling the variability. The goal is to identify and analyse the features that have a persistent nature and derive the required persistent features from them. Next, we showed how the corresponding variable data model could be defined from this persistent perspective. For this, we introduced the concept of annotations into the data model to mark data entities and attributes with variability information.

Chapter 8

The Quiz Product Line Case

In this chapter, we illustrate with a non-trivial example the use of the Feature Assembly Modelling technique to model variable software, a Quiz Product line. The aim of this chapter is to show in detail the process of modelling a complete example for an SPL. The example also shows that the use of perspectives helps scaling down the complexity; it also helps in identifying the features that make up the product line. The Quiz Product Line (QPL) is a family of applications to create web-based interactive quizzes. The Quiz product line is a data centric variable application that is mainly driven by the different capabilities of each individual Quiz creation product. This example demonstrates the use of the Feature Assembly Modelling technique in combination with a centralized data model approach (EER model). The presented Quiz Product Line contains 246 features defined in four perspectives (System, Graphical User Interface, Functional, and Users), and holds 45 different feature dependencies. By this non-trivial example we demonstrate the modelling of a relatively large and complex system using different perspectives. The example also shows that the features of one perspective have a common purpose, which on the one hand makes it easy to spot and identify features, on the other hand allows making a clear mental model of the SPL's capabilities based on the different viewpoints considered.

8.1 Problem Statement

With the Quiz Product Line (QPL) we want to have the possibility to make different types of “quiz creation” applications in order to meet the needs of multiple customers and markets. The product line should be defined such that it allows creating a variable range of products with various capabilities that satisfy a wide range of potential customers. Customers range from customers with simple requirements that only need to use the tool in a single user mode for creating simple text-based quizzes, to expert customers that require a multi user mode that allows them to create more advanced quizzes. It should be able to localize the Quiz applications to meet the requirements of users belonging to different regions or countries; therefore it should have support for several languages namely: English, French, Dutch, Danish, Arabic and Chinese.

The QPL should have support for the creation of four different types of quizzes, namely *Simple Quiz*, *Quiz*, *Exam*, and *Self Assessments*. Furthermore, a *Generate Certificate* feature should be provided for customers who wish to use the software for creating exams and provide a certificate for the exam takers at the end of the exam. This is done via displaying the certificate at the end of the exam; the certificate states the exam taker's final score and displays his/her status (i.e. fail or success). In addition to being displayed on the screen, the generated certificate can also be sent to a specific email address or sent to a printer.

Additionally, to meet the expectations of as many customers as possible, the QPL supports three different quiz publishing possibilities, namely publishing the generated quiz to a *CD*, publishing the generated quiz in *Flash* format, and publishing the generated quiz into *HTML*, the two last methods are used for the creation of online quizzes. There are two possibilities for licencing the quiz creation application, either a single user licence, or a multi user license. In a single user license, the quiz creation application does not store information about the usage of the quizzes. The single user license is intended for these customers that need to create simple text-based quizzes; the generated quizzes are always published via Flash, no other publishing method is supported. For more advanced usage, a multi user license is supported. A multi user license is provided for customers that wish to use the quiz creation application to generate quizzes taken by one or more users. A multi user license allows QPL products to contain one or more of the following quiz types: *Quiz*, *Self-Assessment*, or *Exam*. Additionally, a multi user license calls for a reporting facility that allows the quiz creator(s) to view different *Statistics* concerning the users taking the quizzes. The supported statistics include *Question Usage Statistics*, *User Statistics*, and *Answer Statistics*. Additionally, for each use of the software in Self-Assessment mode a report that states the details of this usage is generated so that users can refer to their self-assessment experience offline. Additionally, while in Exam mode the software also generates a report for each exam take, the generated report is for user test tracking while in multi user licence mode.

As already indicated, the QPL is intended for use by many different types of customers, therefore for enabling the creations of quizzes for example for Math, Physics, and Chemistry an *Equation Editor* feature is available for those customers who require it. The Equation Editor allows integrating math and science symbols into the questions. Also, for more flexibility, two types of Question editors are supported: a *Simple Editor* for those customers that only require the creation of text-based quizzes and a *Rich Editor* which enables the creation of more advanced quizzes that contain rich media such as images, audio and video.

Furthermore, the QPL provides a range of question types such as: *Multiple Choice*, *Fill in the Blank*, *Matching*, *True/False*, and *Sequencing*. Two types of *Multiple Choice* questions are supported, *Single Answer Multiple Choice* and *Multiple Answer Multiple Choice*. Additionally, the question types range from simple text-based questions to questions that contain *Media* such as *Images*, *Audio*, and *Video* for creating more advanced quizzes. The product line should provide different possibilities for the layout of the quizzes and the questions. In the QPL three different quiz layout possibilities are specified: *Simple Layout* for creating simple quizzes; *Template Based Layout* which allows to select a template from a pool of existing templates; and *Custom Layout* for the creation of more advanced quizzes in which the layout is customized by the customers to meet their needs.

The QPL should support the following optional capabilities:

- Defining a passing score
- Defining the final score
- Defining a feedback in case of pass and failure
- Defining a question display scheme, schemes supported by the system are *one per page* or *N per page (i.e. per screen)*; *N* is defined by the users.
- A termination page could exist to show the final score or some feedback to participants.
- In case multi language is supported there should be a facility to allow users to choose the language of the quiz. This feature is only supported in multi-licence versions of the application.

- Multiple types of navigation options may be supported such as *Forward*, *Backwards* or *Question List*.
- A quiz element builder should allow customers to create their quizzes, optionally the quizzes may be generated via selecting the questions from a pool of previously defined questions; this feature is only supported in multi-licence versions of the application.

Additionally, the QPL should support the following optional capabilities for identifying and manipulating questions belonging to a specific quiz:

- Quiz question creation should be supported based on existing question templates for the different question types. The question template indicates how the question and its possible answers (if available) should be laid out on the screen. The theme colour for the questions is based on the associated quiz template used.
- Quiz question editing should be supported.
- Quiz question deletion should be supported.
- Questions options should be supported such as *Correct Feedback*, *Incorrect Feedback*, and *Question Timeout*. Simple quizzes should not contain these features.
- There exist two methods to calculate the score, a fixed score method which is the default and a weighted sum method which is optional.

8.2 Feature Assembly Models for the QPL

In this section, we show the necessary steps taken to model the above mentioned software product line using the Feature Assembly Modelling technique following the guidelines defined in chapter 6. We start by analysing the basic capabilities of the application (by adopting the variability analysis⁴² method described in section 6.2), a Quiz application contains the following set of main (system) features namely: *Questions*, *Layout*, *License*, *Report Generator*, *Operation Mode*, *Question Editor*, *Quiz Question Generator*, *Utilities*, and *Publish*.

Using the Feature Assembly Perspective Selection process (described in section 6.4.7) the following perspectives were identified: *System* perspective; which provided an overview of the required application; *Functional* perspective, which provides an overview of the required variable functionality; *Graphical User Interface* Perspective, which identifies how the system interface varies based on the variability of the functional and user requirements; *Persistent* perspective, which states the data to be stored in the database and how it varies based on the variable functionality; and *Users* perspective to identified the potential users. In the next sections we describe each perspective and indicate how the features are identified to build up the QPL feature assembly models. A logical starting point for identifying the features is the System perspective which captures the main components and capabilities of the QPL.

⁴² In this example the domain analysis is straightforward and given in the problem statement. The nouns identifying (or related) to QPL features are identified in italic in the problem statement.

8.2.1 QPL System Perspective

The System perspective should contain all the main features of the QPL, indicating how they contribute to the variability of the system. The features should be abstract enough (i.e. not too many details) to provide an overview of the system capabilities to all involved stakeholders. The System perspective should not contain too many details of its features; it is only intended to be a starting point to initiate a more in depth modelling of the system by the other perspectives.

Going back to the problem statement, each quiz creation application is composed of a set of **mandatory** features, namely: *Questions*, *Layout*, *License*, *Operation Mode*, *Score*, and *Question Editor*. In addition, a quiz creation application is **optionally** composed of the following features: *Quiz Question Generator*, *Utilities*, *Reports*, and *Publish*. Figure 8.1 shows the complete Feature Assembly model for the System perspective.

The problem statement mentioned several types of question to be supported by the QPL applications. Therefore, the *Questions* feature is an abstract feature (i.e. variation point), which has the following concrete option features (i.e. variants): *Multiple Choice Single Answer*, *Multiple Choice Multi Answer*, *Fill in the Blank*, *Matching*, *True/False*, and *Sequencing*. In any valid product (i.e. Quiz creation application) at least two and there is no upper limit for the selection of these options (i.e. ‘any’); this is specified by the cardinality 2:-. In addition, the questions could be associated with some multimedia; to represent this we define the abstract feature *Question Multimedia*. Question Multimedia has three concrete option features namely: *Audio*, *Video* and *Image*. At least one should be selected and at most three, therefore we define a minimum cardinality of one and a maximum of three (as shown in figure 8.1).

There are four different types of operation modes namely: *Simple Quiz*, *Quiz*, *Exam*, and *Self Assessments*. Therefore we define *Operation Mode* as an abstract feature that is associated with four option features namely *Simple OM*, *Quiz OM*, *Exam OM* and *Self-Assessment OM*, each one of them is a concrete feature. Any valid quiz creation application should have at least one operation mode therefore we define a minimum cardinality of one. The maximum cardinality is set to 4, which means that the maximum number of operation modes allowed in any valid application is equal to the number of available option features. This is indicated by the cardinality “1:4” (as shown in figure 8.1). Furthermore, the QPL supports three different quiz layouts represented as option features of the abstract feature *Layout*; these are *Simple*, *Template Based* and *Custom*. They have a minimum cardinality of 1 and a maximum of 3 (as shown in figure 8.1). In addition, there are two types of licences available, therefore we define *License* as an abstract feature that has two variants *Single User* and *Multi User*, and both are concrete features.

Furthermore, as described by the problem statement, the type of questions supported and the types of operation modes supported influence by each other. Therefore we identify the following feature dependencies which capture these dependencies:

- Simple OM excludes Matching
- Simple OM excludes Fill the Blank
- Simple OM excludes Sequencing
- Self-Assessment OM excludes Single User
- Exam OM excludes Single User

The System perspective also holds features related to the quiz generation process. The *Quiz Question Generator* feature is a concrete feature composed of a *Randomize* feature which is a concrete feature responsible for making the questions random. The feature *Randomize* is

composed of a mandatory concrete feature *Fixed Options* (which represents a normal random number generator) and an optional concrete feature *Branching Path* (which allows creating paths for selecting the next question to display). There is also the *Question Editor* feature which is an abstract features specified into two concrete features *Simple Editor* and *Rich Editor*. In a valid configuration at least one editor type is supported, therefore the abstract feature *Question Editor* has a minimum cardinality of one. Furthermore, the *Score* feature captures the different techniques supported for calculating the quiz score. *Score* is an abstract feature specified by two concrete features: *Weighted* and *Fixed*. *Score* has a minimum cardinality of one.

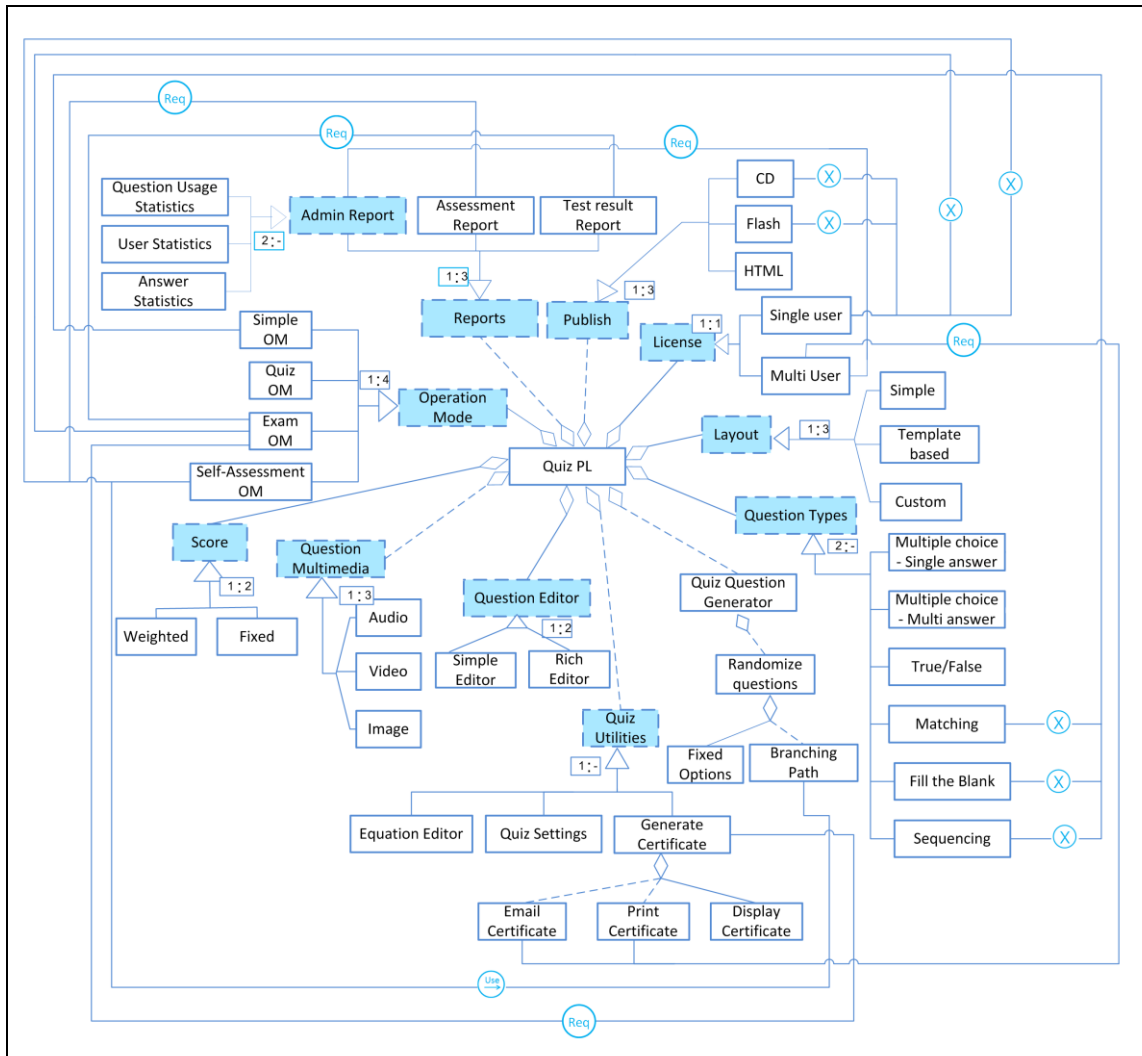


Figure 8.1: QPL System Perspective

The QPL also has a set of advanced features such as the concrete feature *Equation Editor*, the concrete feature *Quiz Settings*, and the concrete feature *Generate Certificate*. These features, although not particularly providing variation of the same functionality, provide a variation of the same concept, namely they represent different variations of utilities. Therefore we specify these features as variants of the *Utilities* feature. There is no obligation to select any of these utilities to be included in specific product. Therefore the *Utilities* feature has an optional feature composition relation with its parent feature (i.e. Quiz). On the other hand, when selected at least one utility should be selected. Therefore, we define a minimum cardinality of one and there is no upper limit on the maximum number of utility features selected therefore we set a maximum of any for the *Utilities* Feature. Furthermore, the

Generate Certificate feature can be further decomposed into *Display Certificate* which is a mandatory concrete feature, *Print Certificate* which is an optional concrete feature, and *Email Certificate* which is also an optional concrete feature.

The System perspective also provides some information on the support provided for publishing the quizzes and the different types of reports supported via two abstract features *Publish* and *Reports* respectively. The *Publish* feature has the following variants (all of which are concrete features): *CD*, *Flash* and *HTML*. At least one specific type of publishing method needs to be selected therefore a minimum cardinality of one is specified. The *Reports* feature is further specified to the following features: *Assessment Report*, *Test Result Report*, and *Admin Report*. We define a minimum cardinality of one and a maximum of three for the *Reports* feature. The features *Test Result Report* and *Admin Report* are concrete features. The feature *Assessment Report* is an abstract feature than can be further specified to *Question Usage Statistics*, *User Statistics*, and *Answer Statistics*. A minimum cardinality of two and a maximum of any is defined for the *Admin Reports* feature (as shown in figure 8.1). The existence or absence of the *Publish* and *Reports* features in a valid product is influenced by other features of the QPL. We model this influence via the following feature dependencies:

- Self-Assessment OM requires Assessment Report
- Multi User requires Admin Report
- Exam OM requires Test Result Report
- Single User excludes Flash
- Single User excludes CD
- Simple OM requires HTML

The next step is to analyse the overall System perspective model to define missing feature dependencies, the dependencies are defined based on the restrictions defined in the problem statement. The following dependencies were defined:

- Exam OM requires Generate Certificate
- Self-Assessment uses Branching path
- Email Certificate requires Multi User

Figure 7.1 shows the complete Feature Assembly model for the System perspective; the feature dependencies are also displayed in the model. As the model grows and the number of feature dependencies becomes too large, we could use the textual specification for specifying them.

Next, we will model the Users perspective to have an overview of the potential users of a quiz creation application; we show the model in the next section.

8.2.2 QPL Users Perspective

As already mentioned, the Users perspective is intended to provide an overview of the potential users of the quiz creation applications. We define the following main features: *Education*, *Business*, and *General*. The three features are considered specifications of the abstract concept *Usage Domains*. The *Usage Domains* feature has a minimum cardinality of one and a maximum of one, which means that only one usage domain should be taken into consideration for a specific product, i.e. a quiz creation application can be configured for only one specific user group.

Education represents the concept of the education domain users. It is modelled as an abstract feature and is further specified by three concrete features namely: *Primary School*,

Higher Education, and *Secondary Schools*. There is no obligation on the upper limit selection of these user groups and therefore the *Education* feature has a minimum cardinality of one and a maximum of any. *Business* is also abstract and is further specified into the following concrete features: *Small-Medium Business*, and *Cooperate Business*. The *Business* feature has a minimum cardinality of one and a maximum of any.

We also define the following feature dependencies, which limit the possible configurations of the QPL for the Education user group (as shown in figure 8.2):

- Secondary Schools excludes Higher Education
- Primary Schools excludes Higher Education
- Education excludes Business

Having modelled the features of the System perspective and the Users perspective, the next step is to proceed with defining feature models for the other perspectives. In principle this can be done in parallel by different teams. In our demonstration, we will proceed with the functional perspective, then the Graphical User Interface perspective, and finally give the Persistent perspective.

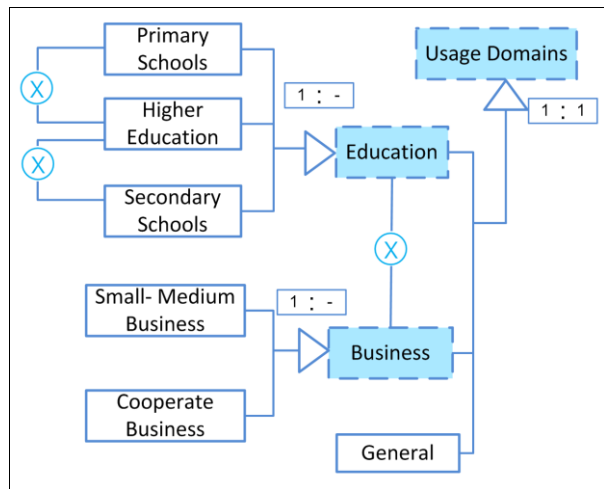


Figure 8.2: QPL User Perspective

8.2.3 QPL Functional Perspective

As already mentioned, the Functional perspective focuses on defining features that represent the functionality provided by the QPL and how this impacts variability. Based on the problem statement, we identify the following main functional features: *Quiz Manipulation*, *Quiz Reporting*, *Quiz Presentation*, *Answer Validation*, *Operation Mode*, *Quiz Settings*, *Operational Settings*, and *Quiz-Question Assignment*. They are explained further on. Figure 8.3 shows an excerpt⁴³ of the Functional perspective feature model. Figures 8.4, 8.5, 8.6, and 8.7 show details of the rest of the model.

In order to capture the allowed possibilities of manipulating a quiz and how they may differ in different products, we define the *Quiz Manipulation* feature. The *Quiz Manipulation* feature refers to the functionalities available to make up a specific quiz so it needs to be present in all products; therefore it is a mandatory feature. In all products of the QPL it should be allowed to create and delete a specific quiz instance. Furthermore, editing a specific quiz instance is only allowed in the case of a *Multi-User License*. Therefore, the *Quiz Manipulation* feature is mandatory composed of *Creation* and *Deletion*; and optionally composed of *Editing*. Furthermore, the dependency *Editing requires Multi User* holds. The *Creation* feature is further specified into four different varieties depending on the specific type of quiz that is created, i.e. Standard Quiz, Exam, Self-Assessment, and Simple Quiz, which map to the features *StdQ Creation*, *Exam Creation*, *SlfA Creation* and *SimQ Creation* as shown in figure 8.3. Each of

⁴³ The complete model was broken up due to space limitations, the rest of the model is presented in figures 8.4, 8.5, 8.6 and 8.7.

these option features is a concrete feature. There is no need for further decomposition of these concrete (option) features because their decomposition is purely functional and does not contribute to the variability of the system. The *Creation* feature has a minimum cardinality of 1 and a maximum of any. Similarly, the *Editing* feature has four different varieties, therefore it is represented as an abstract feature that has the following option features: *StdQ Editing*, *Exam Editing*, *SlfA Editing* and *SimQ Editing* which allow editing the following Quiz elements respectively: Standard Quiz, Exam, Self-Assessment, and Simple Quiz. Each of these option features is a concrete feature. For the *Deletion* feature there is no need for further decomposition because it contains no variability; the deletion process for all quizzes is the same. This is shown in figure 8.3.

Furthermore, the functionality of defining different question instances that belong to the created quiz instances is represented by the *Question Writer* feature. This functionality is available for all quiz products and therefore it is a mandatory feature. The *Question Writer* feature is a concrete feature responsible for the authoring (i.e. adding, editing, and deleting) of the question instances of the different question types. Therefore, it is mandatory composed of the following concrete features: *Question Addition*, *Question Editing* and *Question Deletion*, and optionally composed of the concrete feature *Question Settings*. The addition of questions may vary from one quiz product to another; this is shown by the decomposition of *Question Addition* feature. The *Question Addition* feature is composed of the following mandatory features: *Question Type Selection*, *Question Text Addition*, *Answers Addition*, and *Correct Answer Definition*, and optional composed of: *Question Assessment Addition* and *Question Category Assignment*. Furthermore the selection of these optional features is constrained by the following intra-perspective feature dependencies:

- *System.Self-Assessment OM* requires *Question Assessment Addition*
- *System.Multi User* requires *Question Category Assignment*

The assessment addition is a combination of adding the assessment text and adding the assessment media. To represent this, the *Question Assessment Addition* is mandatory composed of the concrete feature *Assessment Text Addition* and optionally composed of the concrete feature *Assessment Media Addition*. Each created question has a set of options that could be further specified; this is indicated by the *Question Settings* feature. The *Question Settings* feature is optionally composed of the concrete features *Correct Feedback*, *Incorrect Feedback*, and *Question Timeout*, which refer to the setting of correct feedback, incorrect feedback, and question timeout respectively. Additionally, the *Question Settings* feature is mandatory decomposed into the abstract feature *Score Calculation*, which identifies the method for the score calculation. The *Score Calculation* feature has two concrete option features *Fixed Score Calculation* and *Weighted Score Calculation* (for weighted score of the questions) and at least one has to be selected. This is shown in figure 8.3.

For each quiz instance created a set of options are available so that the customers can customize their created quizzes. This set of quiz options varies for the Quiz products derived from the QPL. The feature responsible for representing these options is the *Quiz Settings* feature. The *Quiz Settings* feature is a concrete feature mandatory composed of the following concrete features (which all represent setting of the information they represent): *Passing Score*, *Final Score*, *Failing Feedback*, *Passing Feedback*, and *Title*. Additionally, the *Quiz Settings* feature is optionally composed of the abstract features: *Question Display Scheme*, *Navigation Options*, and *Multimedia*. It is also optionally composed of the concrete features: *Termination Page*, *Language*, *Instructions*, and *Display Question List*, as shown in figure 8.3. The *Multimedia* feature has two concrete option features, namely *Sound*, which is responsible for question sound effects and *QImage*, which is responsible for displaying images for the quiz instances (e.g., correct answer image, wrong answer image, quiz termination image). The *Question Display Scheme* feature identifies the possibilities for displaying the questions in a

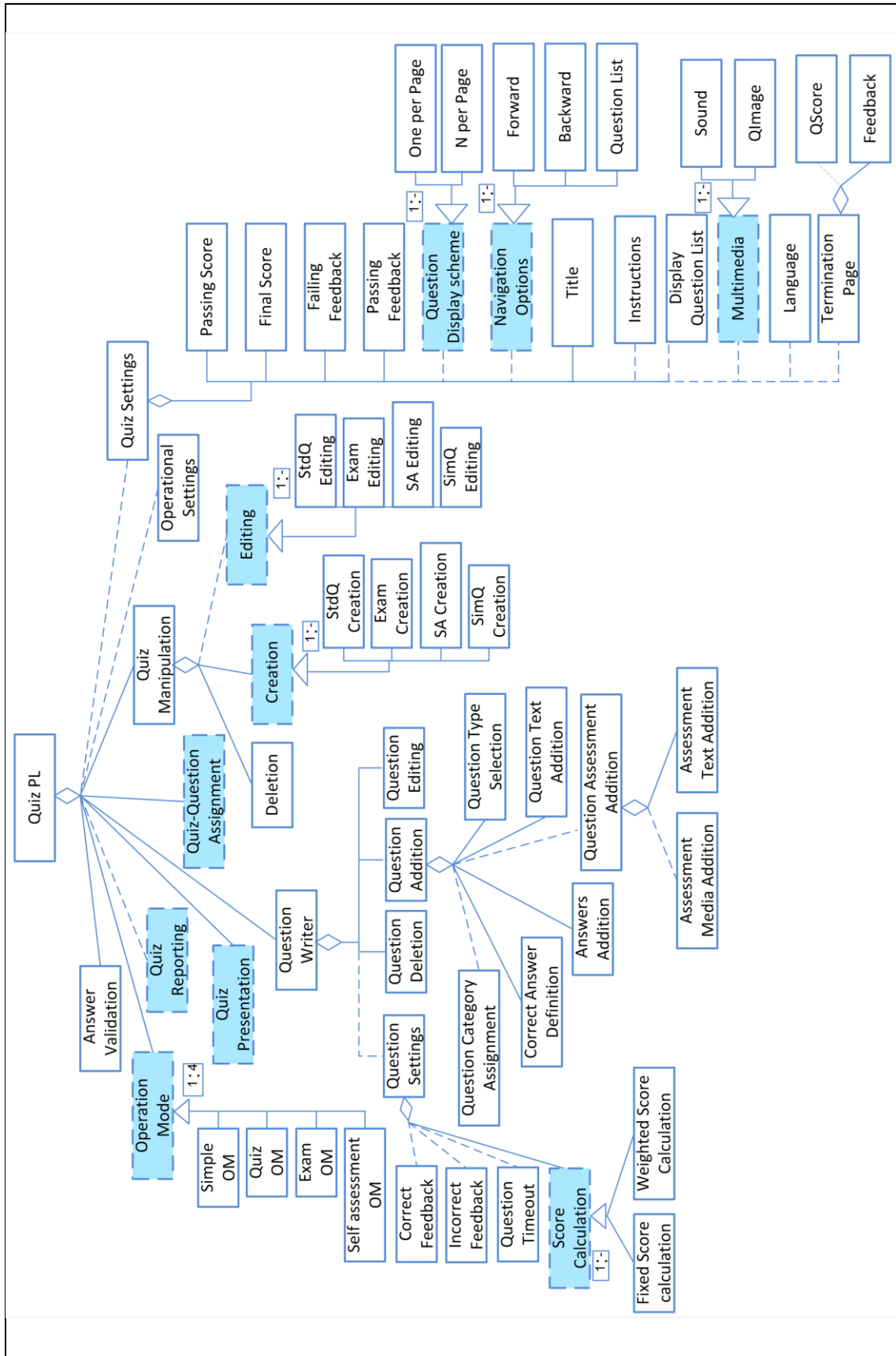


Figure 8.3: An excerpt of the QPL Functional Perspective

certain quiz instance; it has two concrete option features, namely *One per Page* and *N per Page*. The *Navigation Options* has three different navigation possibilities represented by the concrete option features: *Forward*, *Backward* and *Question List*.

The Functional perspective also defines its own *Operation Mode* feature, this feature is the same feature as the one defined in the System perspective. Like in the System perspective, the *Operation Mode* feature is an abstract feature associated with four option features namely *Simple OM*, *Quiz OM*, *Exam OM* and *Self-Assessment OM*, each one of them is a concrete feature, as shown in figure 8.3.

Once a quiz instance is created it needs to be populated with questions, the feature that represents this functionality is the *Quiz-Question Assignment* feature. This is a requirement for all products created by the QPL and therefore it is mandatory (as shown in figure 8.3). There are two possibilities to populate quiz instances with questions 1) through question pooling (e.g., from database) and 2) by fixing the questions into the created quiz instances. A valid Quiz product configuration may have both. Therefore, the *Quiz-Question Assignment* feature is an abstract feature that is specified by the two concrete features *Fixed Assignment* and *Question Pool Assignment*. The *Question Pool Assignment* feature is optionally decomposed into the two concrete features *User Based Assignment* (i.e. Question Pooling based on the user who entered the questions) and *Category Based Assignment* (category based selection of the questions). Furthermore, the feature *Fixed Assignment* should always be selected in all products of the QPL. This is guaranteed by adding the dependency: *Quiz-Question Assignment* requires *Fixed Assignment*.

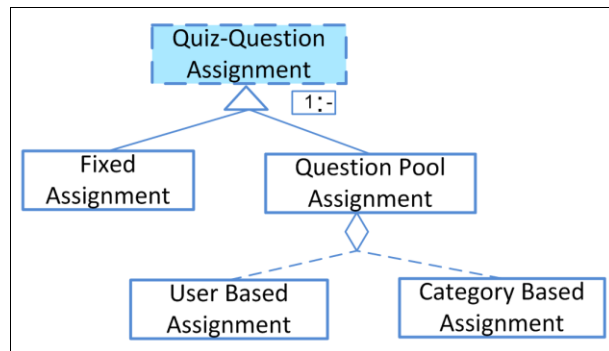


Figure 8.4: Functional Perspective - the Quiz-Question Assignment Feature

The different quiz products created may vary in their purpose of use; therefore the QPL provides flexibility in the usage operation modes that could be required by the different customers. The feature that represents these different modes of operation is the *Operational Settings* feature, which is an optional feature (as shown in figure 8.3). The *Operational Settings* feature is a concrete feature mandatory composed of the abstract feature *Registration* and the concrete feature *Usage Settings*. The feature *Usage Settings* represents the different possibilities that can be customized (by customers) for users taking a certain quiz. It is decomposed into the concrete feature *Question Reviewing* (optional decomposition) and the abstract feature *Result Settings* (mandatory decomposition). The *Result Setting* feature is further specified by two concrete features, *Quiz Result Settings* and *Exam Result Settings*. The *Quiz Result Settings* feature is mandatory composed of the concrete features *Result Display* and *Status Display*. In addition, it is optionally composed of the concrete features *Passing Score Display*, and *Result Storing* and the abstract feature *Result Comparison*. The feature *Result Comparison* has two specifications *Same User Comparison* and *Other Users Comparison*; it has a minimum cardinality of *one*, and a maximum of *two*. The *Exam Result* feature is mandatory composed of the abstract feature *Result Display Scheme*, and the concrete feature *Status Display*. In addition, it is optionally composed of the concrete features *Passing Score Display*, *Certificate Printing* and *Exam Result Storing*. On the other hand, the *Registration* feature is specified by two concrete option features *Optional Registration* and *Enforced Registration*, i.e. the quiz can be configured to force users to register or to leave it up to the user. Figure 8.5 shows the FAM feature model that shows this decomposition.

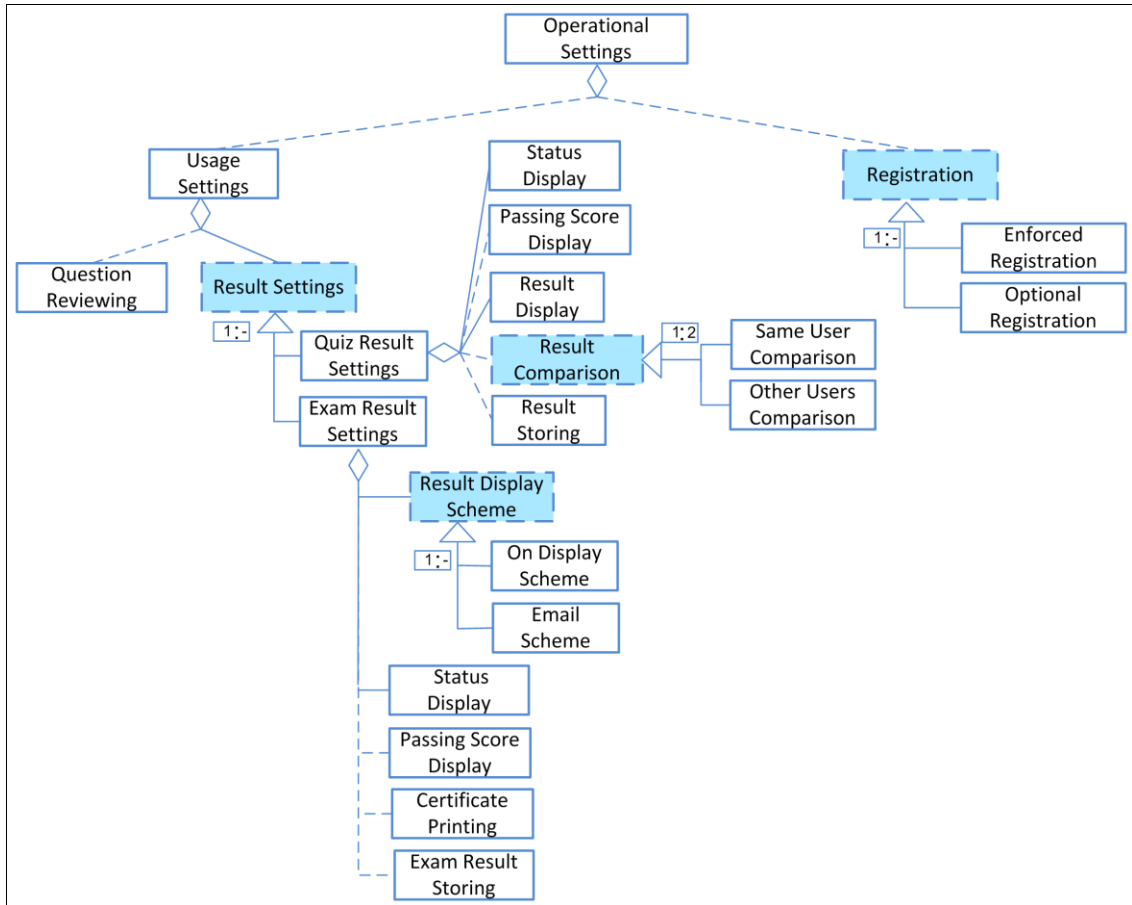


Figure 8.5: Functional Perspective - the Operational Settings Feature

The QPL provides two techniques that allow customers to customize the answer validation process. The *Answer Validation* feature is responsible for defining the possibilities for validating the user answers; as shown in figure 8.6 it is an abstract feature that has two concrete option features, *Instantaneous Validation* and *On Submission Validation*. At least one has to be selected and there is no restriction on the maximum cardinality specified (i.e. any).

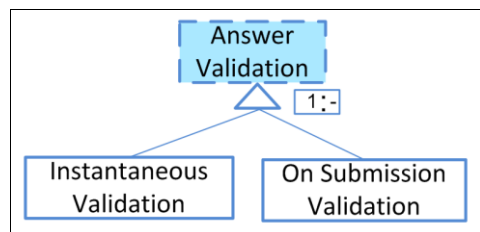


Figure 8.6: Functional Perspective - the Answer Validation Feature

The *Quiz Layout* feature shown in figure 8.7 is a more detailed description of the *Quiz Layout* feature defined in the System perspective. Similar to the one defined in the System perspective, the *Quiz Layout* is an abstract feature, specified by three concrete features namely: *Simple*, *Template Based* and *Custom* which denote three types of supported layouts namely: simple layout, template based layout and custom layout. The *Simple* feature is mandatory composed of the abstract feature *Background Selection* and the concrete feature *Structure Selection*. The *Background Selection* feature has two specifications *Color Customization* and *BGImage Selection*, at least one of them has to be selected. The *Template Based* feature is mandatory composed of the concrete feature *Template Library* which represents a template library of ready to use templates and which allows to select and use a specific template. Additionally, the *Template Based* feature is optionally composed of the concrete features

Template Import and Color Customization. The *Custom* feature is mandatory composed of the concrete feature *Template Editor* and optionally composed of the Concrete feature *Template Designer*.

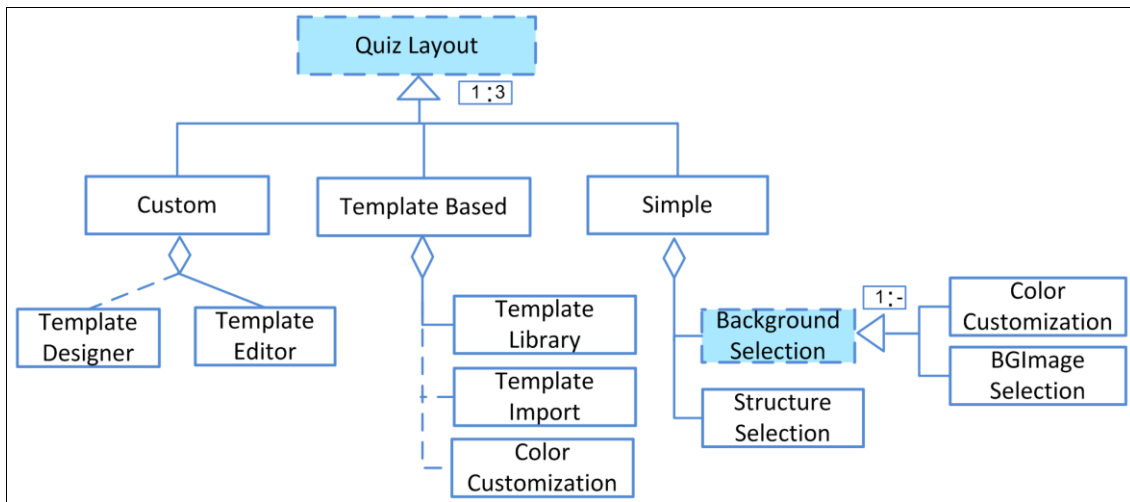


Figure 8.7: Functional Perspective - the Quiz Layout Feature

The QPL also provides different reporting possibilities, these are defined by the *Quiz Reporting* feature shown in figure 8.8. The *Quiz Reporting* feature is an abstract feature that has two abstract option features, *Admin Reporting* and *User Reporting*; there is no obligation on their selection. The *Admin Reporting* feature is specified by the concrete features *Generate Question Usage Statistics*, *Generate User Statistics*, *Generate Answer Statistics*, and *User Comparison Reporting*, in addition to the abstract feature *Custom Reporting*. At least three different report types should be selected, no particular maximum number of features is obligated (i.e. “any”). *Custom Reporting* is further specified by the option features *Exam CR*, *Quiz CR*, *Self-Assessment CR* (where CR stands for custom reporting), all concrete features. The *User Reporting* abstract feature has the following concrete option features: *Exam Result Reporting* and *Assessment Reporting* in addition to the abstract option feature *Quiz Reporting*.

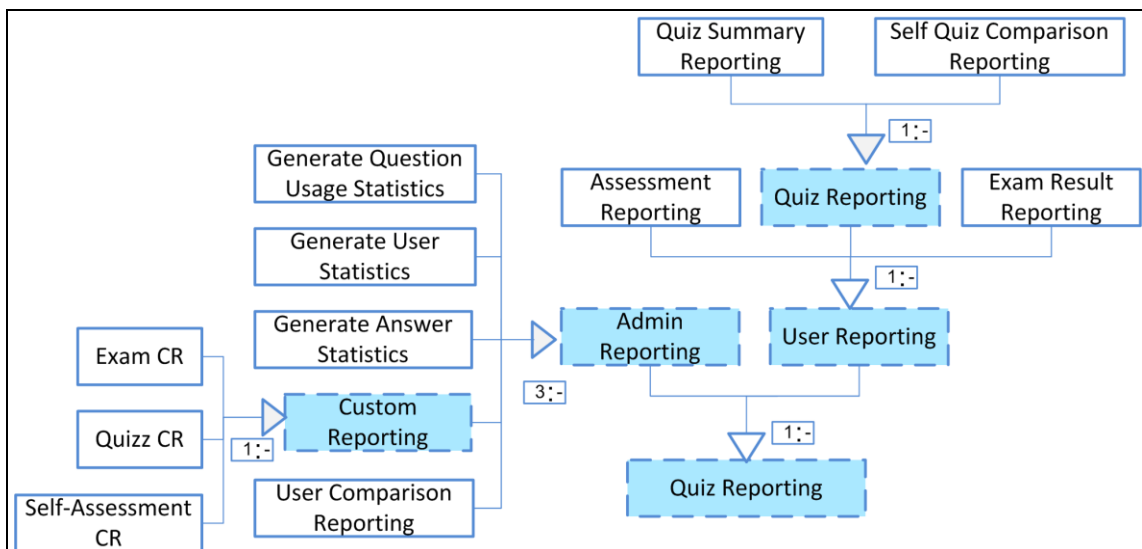


Figure 8.8: Functional Perspective - the Quiz Reporting Feature

The abstract feature *Quiz Reporting* is specified by the concrete option features *Quiz Summary Reporting* and *Self Quiz Comparison Reporting*; at least one has to be selected. Furthermore, the following feature dependencies govern the selection of the reporting features.

- Custom Reporting requires System.Multi License
- Exam CR requires System.Exam OM
- Quiz CR requires System.Quizz OM
- Self-Assessment CR requires System.Self-Assessment OM

8.2.4 QPL Graphical User Interface Perspective

As already mentioned, the Graphical User Interface (GUI) perspective focuses on defining features that the end-user of the application will interact with and view. Figure 8.9 shows the main features of the GUI perspective for the QPL. Some of these features will be further specified in subsequent figures. From a GUI point of view the QPL consists from the following features: *User Reporting*, *Admin Reporting*, *Login*, *Quiz Layout*, *Question Layout*, *Internationalization*, *Registration*, and *Question Pooling*. The features *Quiz Layout*, *Question Layout*, and *Internationalization* are mandatory parts of the QPL GUI, while *User Reports*, *Admin Reports*, *Login* and *Question Pooling* are optional parts of the QPL GUI. The features are explained in more details below.

The feature *Quiz Layout* represents the layout of the quiz from a user interface point of view. It is an abstract feature that has the following concrete option features, *Simple*, *Template Based*, and *Custom* (similar as in the Functional perspective). At least one layout type should be selected. *Template Based* is mandatory composed of the concrete feature *Template Browsing*, and it is optionally composed of the concrete features *Template Preview*, *Template Import*, and *Template Color Customization*.

To support different markets, there is a need to define the user interface elements that are subject to change based on the market requirements; the *Internationalization* feature represents this. The *Internationalization* feature is the feature responsible for providing localization (i.e. adapting the interface to language and customs of different localities). It is mandatory composed of the features *Scrolling*, *UI Components Locality*, and *Text Locality*. *Scrolling* is an abstract feature that is further specified into the abstract features *Vertical* and *Horizontal*. The feature *Vertical* is further specified into the two concrete option features *Top-Down* and *Down-Top*. The feature *Horizontal* is further specified into the two concrete option features *H-RTL* and *H-LTR* which represent the right-to-left and left-to-right directions of the scrolling. The feature *UI Components Locality* is mandatory composed of the abstract feature *UI Direction* and optionally composed of the concrete feature *UI Colors*. *UI Direction* is specified by two concrete option features *LTR* and *RTL* to represent the right-to-left and left-to-right direction of the interface components layout. The *Text Locality* concrete feature is composed the following abstract features: *UI Language*, *Cursor Orientation*, and *Text Direction*. The languages supported by the QPL are represented as concrete features, these are: *English*, *French*, *Dutch*, *Danish*, *Chinese*, and *Arabic*. In a valid product configuration a minimum of two languages should be selected and a maximum of three. The *Text Direction*

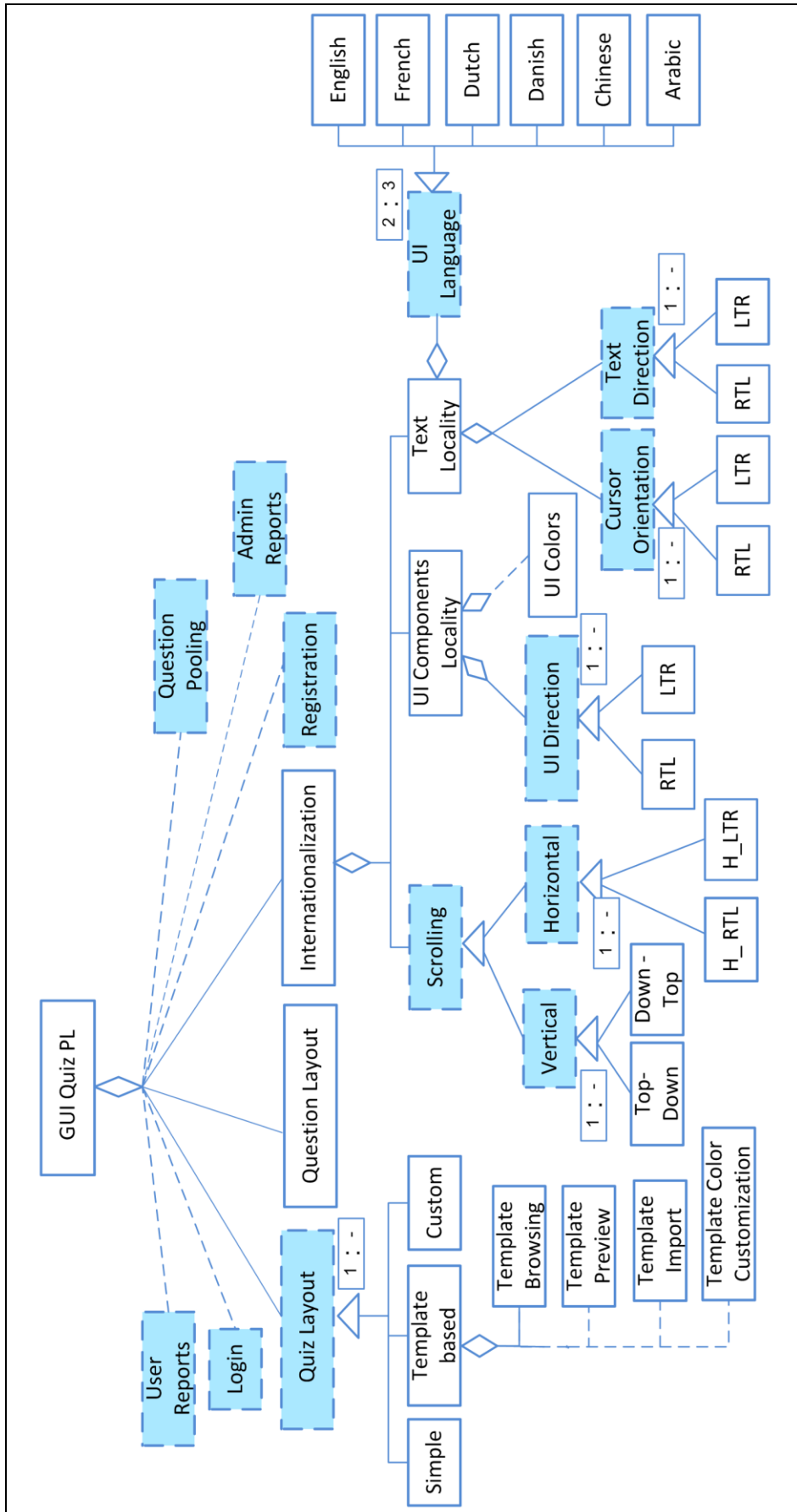


Figure 8.9: GUI Perspective

feature is specified by two concrete option features, *LTR*⁴⁴ and *RTL*. Similarly, *Cursor Orientation* is specified by two concrete option features, *LTR* and *RTL*. Furthermore the following feature dependencies hold:

- English requires *LTR*
- French requires *LTR*
- Dutch requires *LTR*
- Danish requires *LTR*
- Chinese requires *RTL*
- Arabic requires *RTL*
- *RTL* requires *H_RTL*
- *LTR* requires *H_LTR*
- *Language* requires *English*

Allowing users to register, and therefore login, will have an impact of the user interface. Therefore, we define the *Registration* feature and the *Login* feature to represent this characteristic, as they need not be present in all products of the QPL, they are defined as optional. The abstract *Registration* feature, shown in figure 8.10.a, is further specified by the concrete features *User Registration* (which defines a user registration) and *Admin Registration* (which defines an administration registration), a minimum cardinality of one is specified and a maximum of two.

The dependency *Simple OM* excludes *Admin Registration* holds. Similarly there are two types of login defined for the abstract *Login* feature: *User Login* and *Admin Login*; having a minimum cardinality of one and a maximum of two. This is shown in figure 8.10.b.

From a graphical user interface point of view, there are two methods for selecting the questions that belong to a certain quiz created; these are by manual selection or by using a wizard. To indicate this, the *Question Pooling* feature is defined. It is an abstract feature which is further specified by two concrete option features *Question Selection Pooling*, which represents the manual selection of the questions that belong to a certain quiz; and *Question Pooling Wizard*, which represents the use of a wizard to select the questions that belong to a

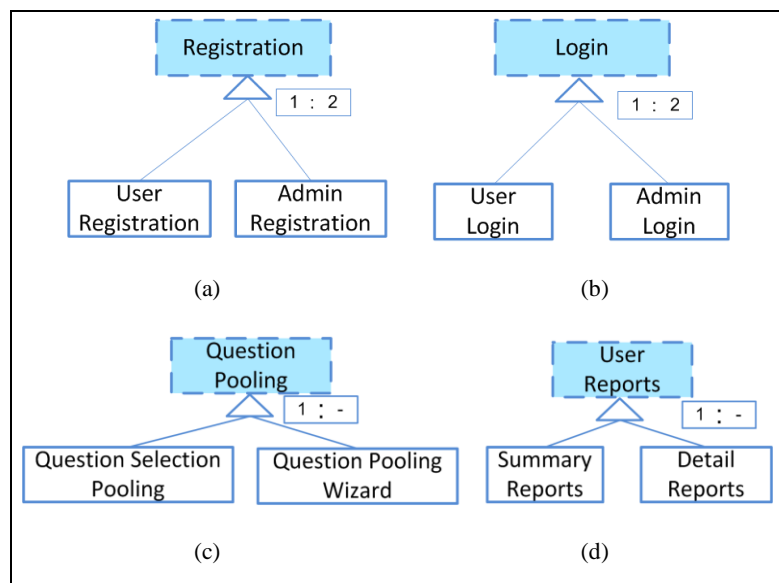


Figure 8.10: GUI Perspective – a) The Registration Feature, b) The Login Feature, c) The Question Pooling Feature, d) The User Reporting Feature

⁴⁴ Note the use of the same feature name to refer to the same feature in the perspective (as mentioned in chapter 6).

certain quiz, see figure 8.10 (c). Similarly, there are two ways to present the user reporting information to end users, either as a non-interactive report listing the information (referred to as summary reports) or as detailed information in which further queries will be issued (referred to as detail reports). This is represented in figure 8.10.d, which shows the *User Reports* abstract feature having two concrete option features *Summary Reports* and *Detail Reports*.

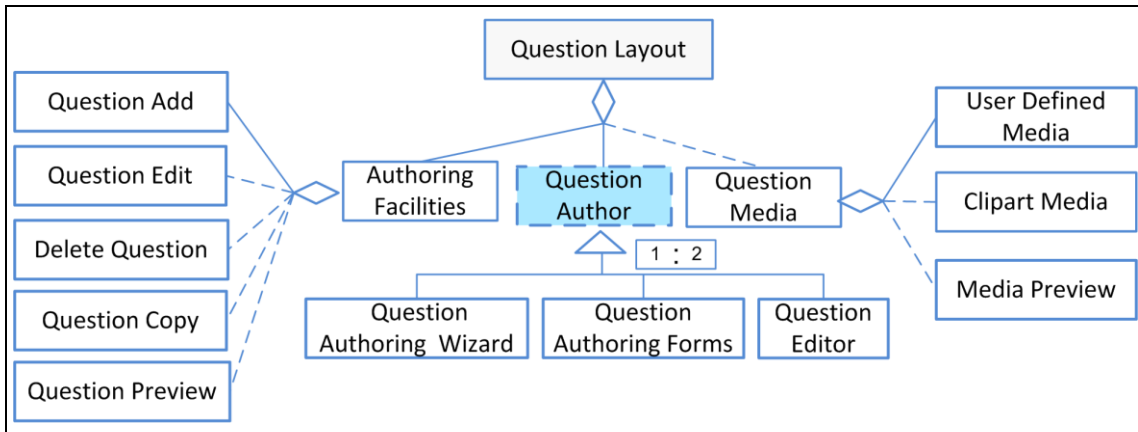


Figure 8.11: GUI Perspective- Quiz Layout Feature

As already mentioned in the Functional perspective, the QPL provides different possibilities for manipulating the questions. This will impact the graphical user interface of the QPL. To represent this, the feature *Question Layout* is defined, its composition is shown in figure 8.11. The *Question Layout* feature is a concrete feature mandatory composed of *Authoring Facilities* and *Question Author*. In addition, it is optionally composed of *Question Media*. The *Authoring Facilities* feature is a concrete feature mandatory composed of the concrete feature *Question Add*. It is optionally composed of the concrete features *Question Edit*, *Question Delete*, *Question Copy*, and *Question Preview*. The *Question Author* feature is an abstract feature that is further specified by the concrete option features that identify different possibilities for authoring a question, these are: *Question Authoring Wizard*, *Question Authoring Forms*, and *Question Editor*; at least one of these features should be selected and at most two. The concrete *Question Media* feature is mandatory composed of the *User Defined Media* feature and is optionally composed of the concrete features *Clipart Media* and *Media Preview*.

8.2.5 Completing the Model

The three previous perspectives identify the features that represent the characteristics, capabilities and appearance of the application. Therefore, if there were no persistent features the modelling process would have stopped here. As already mentioned (in chapter 7), the modelling of the persistent perspective depends greatly on the features identified so far in the different perspectives. For this reason, before proceeding with the persistent perspective, it is recommended to inspect the defined perspectives in order to validate their completeness (i.e. no missing features or dependencies). The model should also be checked for completeness of intra perspective dependencies and feature completeness.

The following dependencies were missing dependencies for the QPL:

- The System feature *Score* is the *same* as the Functional feature *Score Calculation*. Therefore, a same feature dependency should be added as follows:

- `System.Score` same `Functional.Score` Calculation
- The System feature Operation Mode is the *same* as the Functional feature Operation Mode Therefore, a same feature dependency should be added as follows:
 - `System.Operation Mode` same `Functional.Operation Mode`

Furthermore, when analysing the complete⁴⁵ model the following additional dependencies could be identified:

- The multi user license requires the support for defining and changing the usage settings of the application. Therefore, this triggers a *requires* dependency between the two features *Multi User* (belonging to the System perspective) and *Usage Settings* (belonging to the functional perspective) as follows:
 - `System.Multi User` requires `Functional.Usage Settings`
- The exam mode of the quiz application requires the support for navigation buttons in order to navigate through the exam. In addition to a termination page that states the end of the questions and the result of the exam, there should also be support for failing feedback and passing feedback. To achieve this the following feature dependencies need to be added:
 - `System.Exam` requires `Functional.Navigation Options`
 - `System.Exam` requires `Functional.Passing Feedback`
 - `System.Exam` requires `Functional.Failing Feedback`
 - `System.Exam` requires `Functional.Termination Page`
- To prohibit that a valid quiz product allows the inclusion of the questions of types fill the blank, matching, and sequencing in the simple operation model version, the following feature dependencies need to be added:
 - `System.Simple OM` excludes `System.Fill the Blank`
 - `System.Simple OM` excludes `System.Matching`
 - `System.Simple OM` excludes `System.SequencingQ`
- In any quiz product the addition of question assessments requires that the *Question Assessment Addition* feature is selected, this is represented by the following feature dependency:
 - `System.Self-Assessment` requires `Functional.Question Assessment Addition`
- The question editing was represented by the *Question Editor* feature defined in the system perspective. In the functional perspective this was represented by the *Question Writer* feature, therefore a *same* feature dependency should be added as follows:
 - `System. Question Editor` same `Functional.Question Writer`
- The *Question Writer* feature defined in the functional perspective is responsible for adding questions to the application. In the GUI perspective different presentation types for question editing were distinguished, represented by the *Question Author* feature. This situation is represented by adding a *uses* dependency as follows:
 - `GUI.Question Author` uses `Functional.Question Writer`

⁴⁵ In a real case scenario, the complete model will be analysed by different stakeholders, therefore this may result in identifying more feature dependencies. This could be based on emerging requirements, feature co-existence requirements, or feature exclusion requirements.

8.2.6 QPL Persistent Perspective

As already mentioned in chapter 7, the Persistent perspective represented the features with a persistent nature. The objective is to indicate how variability affects the data persistency and therefore the underlying data model. As already mentioned, identifying features for the Persistent perspective is driven by the features defined in the other perspectives. For the QPL, we can distinguish three main persistent features: 1) the *Persistent QPL* feature which represents a quiz (along with all its associated questions, media, options ... etc.), 2) the *User* feature which represents a user taking a quiz together with all his information, 3) the *User-Quiz Info* feature which represents the information for a user taking a certain quiz. We will model each of them in more details.

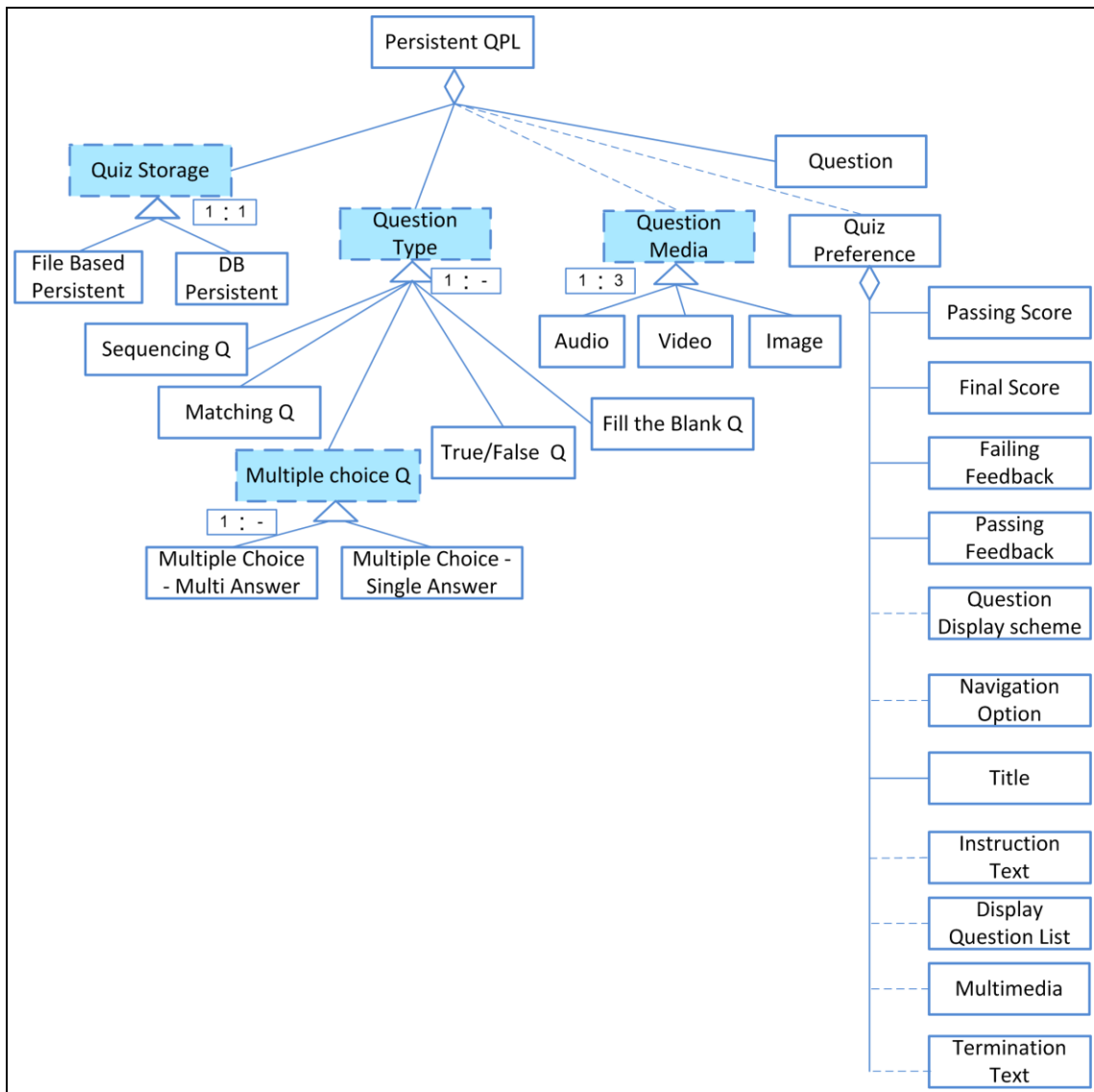


Figure 8.12: Persistent Perspective –Persistent QPL Feature

The concrete *Persistent QPL* (persistent) feature (shown in figure 8.12) is composed of features related to the data stored for generating quizzes within an application. We distinguish two types of persistency, resulting into the features *File based Persistent*, and *DB Persistent*. Both are specifications of the abstract feature *Quiz Storage*. Furthermore, the System

perspective identifies different types of questions supported by the QPL and the Functional perspective defines a set of options available for each quiz (represented by the *Quiz Settings* feature). Based on this information, we can conclude that the *Persistent QPL* feature should be mandatory composed of the abstract feature *Question Type*, the abstract feature *Quiz Storage*, and the concrete feature *Question*. In addition, the *Persistent QPL* feature is optionally composed of the abstract feature *Question Media* and the concrete feature *Quiz Preference*; this is shown in figure 8.12. Also, the following intra-perspective dependencies hold:

- *System.Simple OM uses File Based Persistent*
- *System.Multi User excludes File Based Persistent*

The abstract feature *Question Type* is specified by the following concrete option features: *Sequencing Q*, *Matching Q*, *True/False Q*, *Fill the Blank Q*, in addition to the abstract feature *Multiple Choice Q*. The feature *Multiple Choice Q* is specified by two concrete option features: *Multiple Choice - Single Answer* and *Multiple Choice - Multi Answer*. In any valid product of the QPL at least one type of question is supported, therefore a minimum cardinality of one is defined.

The abstract feature *Question Media* is specified by the following concrete option features: *Audio*, *Video*, and *Image*, with a minimum cardinality of one and maximum cardinality of three. The *Quiz Preference* feature is used to store the quiz information of the *Quiz Settings* feature defined in the Functional perspective (please refer to section 8.2.3), therefore it is composed of the features that contain data to be stored for a particular quiz.

The *Question* feature represents the persistent information associated with the different questions of a product. The *Question* feature is a concrete feature that is optionally composed of *Question Options* and *Question Assessment*. The *Question Assessment* feature is mandatory composed of *Assessment Text* and optionally composed of *Assessment Media*. The *Question Options* feature is optionally composed of: *Weight*, *TimeOut*, *Wrong Answer Feedback*, and *Correct Answer Feedback*, as shown in figure 8.13.

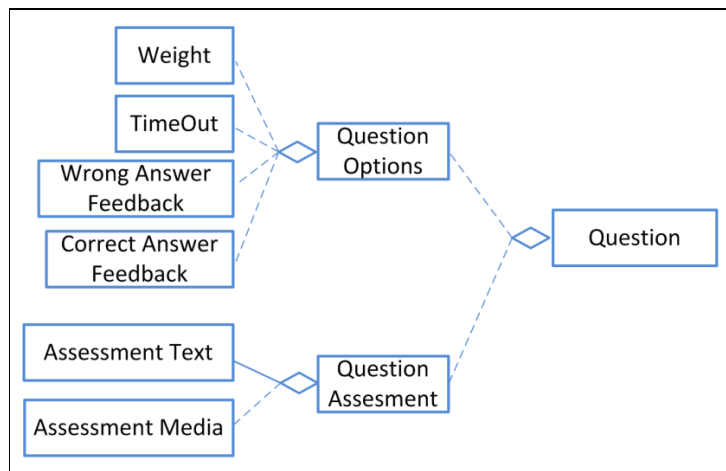


Figure 8.13: Persistent Perspective – Question Persistent Feature

The *User* feature represents the data associated with the end-users of a QPL application. Although the feature “User” (i.e. test taker) was not explicitly present in the other QPL perspectives, user related features exist in the Functional perspective (e.g., *User Login* and *User Registration*). The features composing the *User* feature stem from the System perspective and the Users perspective and they represent the information that should be stored about the users of the application. The Feature Assembly model of the *User* feature is shown in figure 8.14; it is mandatory composed of the *Login Information* feature (which is composed of *User Name* and *Password*), the *Name* feature and the *Email* feature. In addition, it is optionally composed of the *User Details* feature which holds additional information on the users of an application. The *User Details* feature is based on the different user features defined in the Users perspective. It is an abstract feature associated with two concrete option features, *School* and

Business. *School* is mandatory composed of *School Name*, *School Grade*, and *Class*. *Business* is mandatory composed of *Department* and *Branch*.

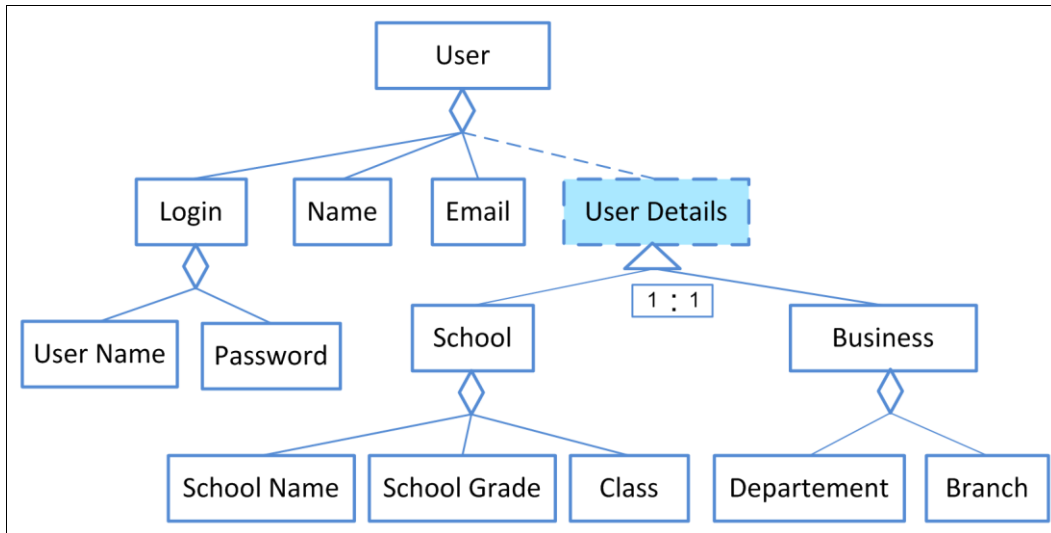


Figure 8.14: Persistent Perspective – User Persistent Feature

Figure 8.15 shows the Feature Assembly model for the feature *User-Quiz Info*, which represents a user having taken a certain quiz or exam. It is an abstract feature that is further specified by two concrete features, *Exam* and *Quiz*. *Exam* is mandatory composed of *User*, *Time stamp*, *Score*, and *Status* optionally composed of *Exam Questions* and *User Answers*. *Quiz* is mandatory composed of *User*, *Time Stamp*, and *Score*. Note that the *User* feature is used for both *Exam* and *Quiz* and also it is the one shown in figure 8.14.

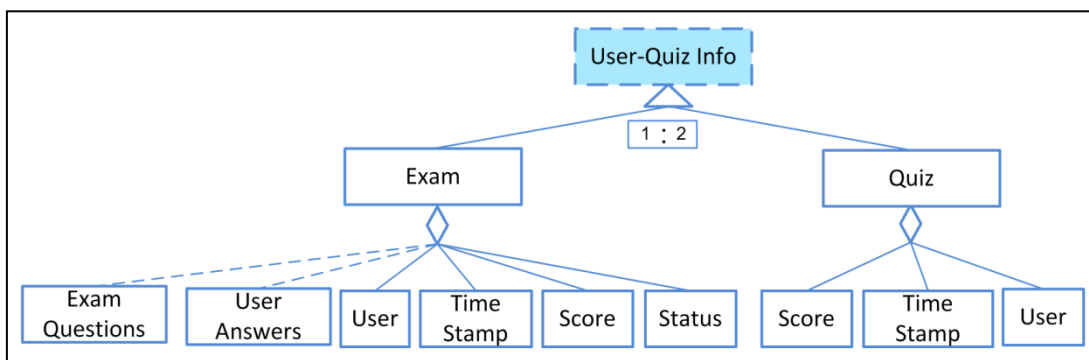


Figure 8.15: Persistent Perspective – User-Quiz Info Persistent Feature

Furthermore, the following set of dependencies show how the Persistent perspective related to the other perspectives.

- System.Muli User requires Persistent.User
- Users.School requires Persistent.School
- Users.Bussiness requires Persistent.Bussiness
- System.Self-Assessment requires Persistent.Question Assessment

8.3 QPL Variable Data Model

As already mentioned in chapter 6, the Persistent perspective is an intermediate step between the variability of the software product line and the variability of the underlying database schema. To obtain the QPL variable data model, we adopt the centralized data model approach described in section 7.2.1. This allows establishing a link between the features defined in the Persistent perspective and the (variable) entities that should be part of the derived variable data model. The mappings used to relate Persistent features to data model entities are shown in listing 8.1. From this mapping we are able to propagate feature variability to entities defined in the data model. The result of this process is the EER data model⁴⁶ for the QPL example shown in figure 8.16

```

o System.Quiz Application <<relates_to>> Data_Model.User
o Persistent.Question <<maps_to>> Data_Model.Question
o Persistent.Passing Feedback <<maps_to>> Data_Model.Passing Feedback
o Persistent.Failing Feedback <<maps_to>> Data_Model.Failing Feedback
o Persistent.Termination Text <<maps_to>> Data_Model.Termination Page
o Persistent.Filltheblank <<maps_to>> Data_Model.FilltheblankQ
o Persistent.Matching <<maps_to>> Data_Model.MatchingQ
o Persistent.Sequencing <<maps_to>> Data_Model.SequencingQ
o Persistent.True/False <<maps_to>> Data_Model.True/False Q
o Persistent.Question Assessment <<relates_to>> Data_Model.Question
  Assessment
o Functional.Navigation Options <<maps_to>> Data_Model.Forward and
  Data_Model.Backward
o System.Multi User <<relates_to>> Data_Model.Admin
o System.Multi User <<relates_to>> Data_Model.Question Options
o System.Multi User <<relates_to>> Data_Model.Quiz Element Options
o Persistent.User <<relates_to>> Data_Model.User
o System.Exam <<relates_to>> Data_Model.Sound Effects

```

Listing 8.1: Feature Assembly-to-data model mappings

⁴⁶ An Example of how this is used for deriving tailored product data schemas is shown in our publication: Towards Modeling Data Variability in Software Product Lines [Abo Zaid and De Troyer, 2011]

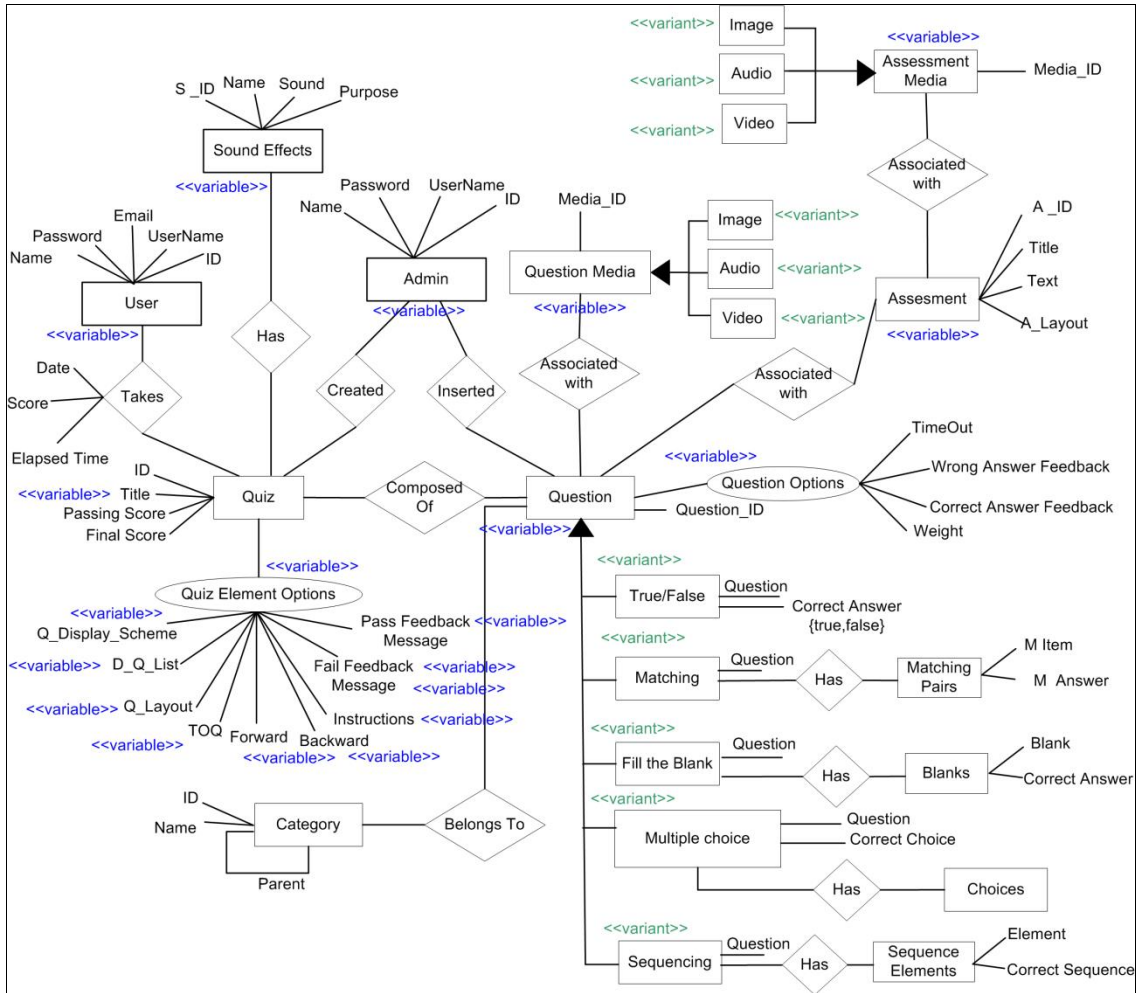


Figure 8.16: QPL Variable Data Model (Represented with EER)

8.4 Extensibility of the Feature Assembly Modelling Technique – An example

As already mentioned, one of the merits of the Feature Assembly Modelling technique is its extensibility. A new perspective can be added to the set of perspectives (defined in section 6.4) if there is a need for it. Defining a new perspective implies defining the purpose for the newly defined perspective (i.e. narrowing down the general perspective definition given in definition 6.2) and providing a definition for the concept of “feature” within the newly defined perspective (by narrowing down the general feature definition given in definition 6.3)

In the QPL there is a need to extend the QPL for different markets and therefore for different cultures. This need was taken into account in the Graphical User Interface where a software localization strategy was adopted. Software localization focuses on the localization of the software aspects such as the interface elements and the interface language. This may not be enough in order to gain a true significant market segment, as the software may remain far from the end user’s anticipation, ideas, and even how he does his tasks. In order to gain a real valuable share of the market, there could be a need for the development of software that meets different cultural values, ideas and procedures (according to Kersten et al. [2002]). This can

only be done if culture⁴⁷ is taken into account in the early development of the software. Cultural aspects should not only affect the interface layer of the software but also penetrate deep in the core; such software is referred to as being culture-dependent. The core of a software artefact embeds decision-making, rules of behaviour and patterns of actions that depend on culture [Kersten et al., 2000].

In order to combine cultural variations with the QPL product variability, we define an additional perspective that represents cultural aspects that affect the QPL and we call it the *Cultural perspective*. The *Cultural Perspective* takes into account the different cultural aspects that may influence the localization of the quiz products when targeting different markets. The features identified in this perspective are based on the understanding of the different cultural dimensions⁴⁸. The cultural dimensions that will affect the localization of the software are thus mapped into features in the Cultural perspective. The importance of this mapping is the influence these “cultural features” could have on some possible combinations of product features. This influence will spread to features belonging to other perspectives and will be denoted via the intra-perspective feature dependencies.

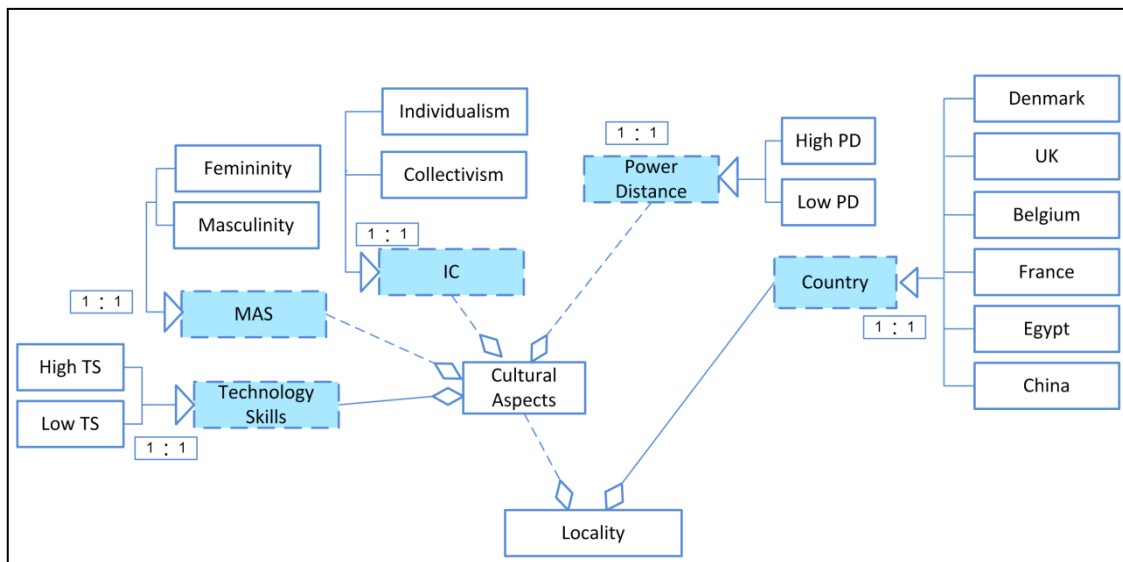


Figure 8.17: Cultural Perspective

Figure 8.17 shows the Feature Assembly model of the Cultural Perspective. The main feature in the model is the *Locality* feature which is a concrete feature mandatory composed of the abstract feature *Country* and optionally composed of the concrete feature *Cultural Aspects*. The *Country* feature is an abstraction of the different target countries; therefore it is specified by the following concrete features *Denmark*, *Belgium*, *France*, *Egypt*, and *China*. At least one country has to be selected and at most one, as specified by the cardinality shown in figure 8.17. The *Cultural Aspects* feature is optionally composed of the abstract features: *Technology Skills*, *MAS*, *IC*, and *Power Distance*. *Technology Skills* is specified by the concrete features *High TS* and *Low TS*. *MAS* (refers to masculinity versus femininity) is specified by the concrete features

⁴⁷ Culture is a world of symbols constructed by people; it is a structure of meanings, beliefs and values that condition human behaviour allowing for its interpretation and purposefulness as defined by [Kersten et al. 2000].

⁴⁸ Cultural dimensions are mostly sociological dimensions, or value constructs, which can be used to describe a specific culture. In the example, we use the cultural dimensions identified by G. Hofstede commonly referred to as Hofstede's Culture Dimensions. For more information we direct the readers to <http://www.geerthofstede.nl/culture/dimensions-of-national-cultures.aspx>.

Masculinity and *Femininity*. *IC* (refers to individualism versus collectivism) is specified by the concrete features *Individualism* and *Collectivism*. *Power Distance* is specified by the concrete features *High PD* and *Low PD*. Furthermore, the following feature dependencies can be identified:

- Belgium requires Dutch
- Belgium requires English
- Belgium requires French
- Egypt requires Arabic
- Egypt requires English
- France requires French
- Denmark requires Danish
- China requires Chinese
- UK requires English
- System.Custom uses Individualism
- High PD requires System.Template Based
- High PD excludes System.Custom
- Collectivism requires System.Standard
- Users.Primary schools uses Individualism
- Layout uses Low TS
- Users.Primary School uses Low TS

8.5 Lessons Learned

Having used the Feature Assembly Modelling technique on this large example we came to the following observations:

1. The Feature Assembly Perspective Selection process (described in section 6.4.7) helps identifying the set of perspectives relevant to model a certain problem. This is a simple approach and yet provides more guidance as opposed to leaving it entirely up to the involved modellers to define the appropriate perspectives which could be efficient for the configuration problem (e.g. the work on using perspectives as abstraction mechanism for configurations driven by feature models (e.g. Hubaux et al. [2011], Schroeter et al. [2012], Acher et al. [2012])) but maybe less suitable for modelling.
2. Starting with the system perspective helps revealing the key capabilities and characteristics of the system, which are then investigated in more details in the other different perspectives. The system perspective as an abstraction mechanism helps moving from more abstract system commonality and variability to more concrete and detailed ones in subsequent perspectives.
3. Features may be defined based on more than one point of view. Furthermore a feature may have more than one face. Therefore, representing features using the multiple perspective approach helps projecting the feature on the different concerns that it relates to, allowing to focus on one concern at a time.

4. The modelling process stops when no variability is encountered when decomposing features (as already mentioned in section 6.5.2).

Some comments on the usability of the technique are:

1. The Feature Assembly Modelling language has strict rules for modelling relations between features. We found this helpful in defining better models and it forced us to have a thorough understanding of how the different features are composed/decomposed, how they relate to variability, and affect each other.
2. Change and maintenance of the models already created, due to emerging requirements is possible. Adding new features to a perspective requires an understanding of how the added features would influence the already existing features. This influence (if any) is then expressed by feature dependencies.
3. The technique allows using only those perspectives that are useful for identifying features that characterize and describe the capabilities of the product line. So, modellers are not forced to make models that they don't need or don't consider useful.
4. The extensibility of the technique allows adding perspectives when they are required.
5. By separating feature composition from feature specification/generalization the Feature Assembly approach helps defining the features of the product line focusing on either feature composition or feature specification. This has resulted in easier to spot variation points and a better understanding of the variable capabilities of the product line.

8.6 Summary

In this chapter, we have demonstrated with an example how the Feature Assembly Modelling technique can be used in practice. We have presented the Quiz Product Line example, which is a product line for delivering question based quizzes and exams. We have identified the following perspectives *System* perspective, *Users* perspective, *Functional* perspective, *Graphical User Interface* Perspective, and *Persistent* perspective. For each of these perspectives we have identified the features that belong to it and their feature-to-feature dependencies. Furthermore, we have shown how a variable data model can be derived from the Persistent perspective using the techniques described in chapter 7.

Additionally, we showed with an example how the Feature Assembly Modelling technique is extensible. We have added a new perspective, the *Cultural* perspective, to the set of standard perspectives. We showed how the features of the new perspective affected the overall models of the QPL. This effect was captured via a set of feature-to-features dependencies. Finally, we concluded the chapter with a set of conclusions about the use of the Feature Assembly Modelling technique for this example.

Chapter 9

The Feature Assembly Reuse Framework

In this chapter, we present the Feature Assembly Reuse Framework which allows modelling variable software by combining both *reusability* and *variability*. The Feature Assembly Reuse Framework is a *modelling by reuse* method for creating Feature Assembly models. It allows reusing model constructs (i.e. features) of previously defined variable software products while also defining new ones. The idea that Feature Assembly brings is the ability to *reuse domain analysis information* in new projects. We believe that software reuse should start at a design level. In doing so, reuse could be promoted and supported from the initial software conception phase (i.e. domain analysis) through the complete software development life cycle. In order to do so the Feature Assembly Reuse Framework proposes to specify variable software products (in terms of Feature Assembly models) by combining and reusing (existing) software features accompanied with some of their relations and dependencies. In the previous chapters, we have shown how the Feature Assembly Modelling technique supports the modelling of well-structured and scalable feature models. In this chapter, we focus on the *feature reusability* aspect provided by the Feature Assembly approach.

The Feature Assembly Reuse Framework answers our second research question RQ2, particularly, RQ2.3, “*How can the principle of modelling with reuse be introduced to feature modelling*”, by introducing feature reuse as part of the Feature Assembly modelling practice. This involves two steps, firstly making the features available for reuse and secondly actually creating new feature models by reusing these features. Therefore, once a product line is defined by specifying its Feature Assembly models, the features in these models (together with their product independent information) are stored in a “Feature Pool” for later reuse. This pool of features allows for creating different product (lines) by reusing the features. In addition, whenever an existing product line undergoes a change in its scope or requirements or a new product is needed, new features can be introduced and added to the pool allowing it to continuously grow and act as a repository of reusable features. In the next section, we will discuss the motivation behind the Feature Assembly approach. Next, we explain the Feature Assembly Reuse Framework for creating Feature Assembly Models by means of reuse. Finally, we will conclude the chapter with an example based on the Quiz Product Line presented in chapter 8.

9.1 Why Feature Assembly?

Feature Assembly is based on the principle of assembling products by assembling parts, known in industry as assembly lines. The introduction of assembly lines in industry has paved the way for mass production and mass customization leading to the high productivity of today’s industry. Using the same principle in software development, mass production could be achieved via *reusability* while mass customization could be achieved by introducing *variability*.

The physical parts that compose the final product could correspond to *software features* in software. Similar as for physical parts, varieties of a software feature may be possible. These different varieties can be used for customizing the software products. It can be observed that within a certain domain, many features are common and can be reused in more than one piece of software. As an example, a *spelling checker* can be used in many different *office* products. This observation makes Feature Assembly feasible in practice and yet even favourable. Business-wise, this reuse of features could increase productivity and decrease the development cost if anticipated from the start. Important to notice is that we aim for reuse of the software features, and partial reuse of feature models (i.e. reuse of design artefacts). This reuse of features will also propagate to reuse of the software artefacts that actually realize these features (e.g., libraries, components, templates, classes, etc.), in this way enhancing the overall software reuse process.

The principle of assembling a certain product from pre-existing artefacts has previously been proposed in software engineering. However, in general the assembly is situated at the code level. For example, Wang [2000] states “*One of the essential characteristics of engineering disciplines is to build a product by assembling pre-made, standard components*”. When we examine the reuse experience in Component Based Development (CBD) [Wang, 2000] [Heineman and Councill, 2001], we see that CBD is based on developing software by composing pre-existing components. Furthermore, there is a separation between the development of the components and the development of the software that will utilize these components [Crnkovic et al., 2006]. This has called for creating self-contained components that would then minimize the writing of code to only gluing code (code that glues the components together). Although the idea of CBD did not achieve its merits in software development in general, it has been a great success in some specific domains. For example (web) service-based applications [Srivastava and Koehler, 2003] [Dustdar and Schreiner, 2005], and e-learning applications [Menéndez and Prieto, 2008] are often built using a CBD approach. The reason for this is that while the products developed are quite diverse, there is a minimum level of commonality between the required capabilities allowing for a good opportunity for reuse. Furthermore, reuse in a single domain is more successful due to the sharing of the same domain knowledge. Therefore, the opportunity of finding reusable components is higher than that of reuse between different domains. For example, in web services, applications are assembled from a set of appropriate web services according to the functionality they provide. Web service discovery and identification plays an important role in the success of the web service composition approaches [Srivastava and Koehler, 2003]. Therefore, web services are annotated with information on their usage; this description is then stored in a central web service registry. To find a certain web service, the registry is inspected [Dustdar and Schreiner, 2005].

Revisiting the software product line development in practice, a software product line often undergoes adjustments to meet the continuous changes in customer and market requirements [Van Ommering and Bosch, 2002]. This results in widening the scope and diversity of the products that the product line delivers. As the product line matures, its scope may significantly widen due to the introduction of new features. This causes on the one hand a decrease in the complete commonality (i.e. the features that are common to all members of the product line), and on the other hand, an increase in the partial commonality (i.e. the features that are common to a subset of the members of the product line) [Van Ommering, 2002]. This scenario often results in setting up a series of product lines rather than one, and is the first scenario for what is sometimes referred to as multiple product lines (see section 3.5 on multiple product lines). Another relevant scenario defined by Bosch [2009] is when the diversity of the product line increase and there is a need for openness to third party development. Bosch referred to this as software ecosystems [Bosch, 2009], in which a company should make its *platform* available for third party development. The focus of these works has been on identifying the required architecture support for realizing such multiple software product lines.

Feature Assembly allows extending the scope of a variable product or a product line (e.g., to add new variants to existing features and add new features to the set of supported features) at a modelling level. This is similar to the above-mentioned scenarios of multiple product lines but focus is on supporting the (conceptual) modelling of evolving to a set of product lines (i.e. multiple product lines). However, Feature Assembly allows to model different product lines (as well as different products) from existing features by supporting reuse of features as early as the domain analysis phase. Therefore, Feature Assembly allows creating a family of product lines which all have clusters of common features. These product lines may have different purposes and yield different applications (i.e. products).

As already mentioned, opportunity for reuse is higher among applications developed for the same domain, as Frakes et al. [1998] state: “*The domain-centred view of software engineering recognizes that most organizations do not build completely new software applications. Rather they build variants of systems within at most a few problem domains, or business areas*”. The same applies for companies developing variable products without systematically adopting a product line technique, for example companies offering customized versions of their products to different customers. Another example is companies delivering customer specific products, these find a large amount of overlap between their different delivered products and often see the need to *productize* their software to a product that holds the necessary variability to be customized to fits the needs of different customers and markets [Artz et al., 2010] [Leenen et al. 2012]. This accompanied with the fact of the continuous increase in the product (line) diversity, as previously mentioned, puts companies in a situation in which they produce different products with a high level of partial commonality. It could even be the case that a company that started with one product line evolves to having multiple related product lines, all of which have some common features but also new ones. These companies have already identified points of variability and commonalities related to the domain, but need to consider new customer requirements. These customer requirements may again have commonalities as well as individual differences. It is important to reuse this information, whenever there is a need for a new product (line).

Furthermore, adopting a reuse strategy at a domain analysis level paves the way for reuse at later stages of the development (e.g., at architecture level and at code level). In this case, we envision that making new products is an issue of assembling previously defined features along with new features. Figure 9.1 shows an overview of this vision, starting with a *Product A*, of which its building blocks (in this case features) are stored for later reuse (in a so-called Feature Pool), *Product B* and *Product C* reuse some of the features of *Product A* and also define their own features which are also stored in the Feature Pool for later reuse, and so on.



Figure 9.1: Overview of the Feature Assembly Process

9.2 Overview of the Feature Assembly Reuse Framework

The Feature Assembly Reuse Framework is a conceptual framework for *modelling with reuse*. It allows modelling variable software by assembling new features as well as previously defined ones. While in traditional software reuse, the focus is on the reuse of physical software artefacts, in Feature Assembly the focus is on reuse of features, i.e. design artefacts. The reuse is about the features and their compositions as well as relations with other features that might also be useful in a reuse setting. For example, a *Questions* feature may be reused in many different applications such as a *Quiz product line (QPL)*, an *Exam product line*, and a *Questionnaire product line*. Therefore, since this feature was already analysed and modelled for the QPL (please refer to sections 8.2.1 and 8.2.3 for the analysis of the *Questions* feature) we can reuse this feature for the Exam, and Questionnaire product lines. The reuse adopted in Feature Assembly is actually a partial reuse of previously defined Feature Assembly models. Note that when we reuse *Questions*, we reuse it together with its option features and some of its dependencies. For example the dependency: *Questions requires Question Authoring* could hold for all applications using the *Questions* feature.

Furthermore, reuse of models paves the way for reuse at the component level. Therefore reusability is considered very early in the development life cycle without actually being an overload in the development process. Taking reusability into account at the design stage is complementary to reuse at the component level and could enhance the reusability of the components.

The Feature Assembly Reuse Framework supports creating Feature Assembly models for variable software (aka software product lines) by assembling features from a continuously growing repository of features, called the “Feature Pool”, as newly defined features are added to the Feature Pool, allowing the pool of features to continuously grow and evolve over time. To achieve this, the Feature Assembly Approach is adopting a hybrid methodology that combines both a *top-down* and *bottom-up* design approach. It is well known that top-down design approaches allow decomposing large systems into smaller parts allowing to better understanding and model each part. For variable software, this decomposition process also includes variability (as already mentioned in chapter 6). In a bottom-up design approach, system parts are put together to build up a larger system. In order to do so, there needs to be some form of awareness on the existence of these parts and knowledge of their capabilities. This also addresses one of the limitations of mainstream feature modelling techniques, i.e. adopting a top-down hierarchical modelling approach (see L1.4, section 5.1.1 for more details)

Figure 9.2 shows an overview of how the Feature Assembly Reuse Framework works. A company⁴⁹ starts with one product (line) *Variable Product 1*, and uses the Feature Assembly Modelling technique to model this product (line). Reusable features will be stored in the Feature Pool. Later on, when a second product (line) is needed, *Variable Product 2*, features from the Feature Pool can be reused.

⁴⁹ Note that the company can also start by developing a software product line (instead of a single product) and later on develop a second product line.

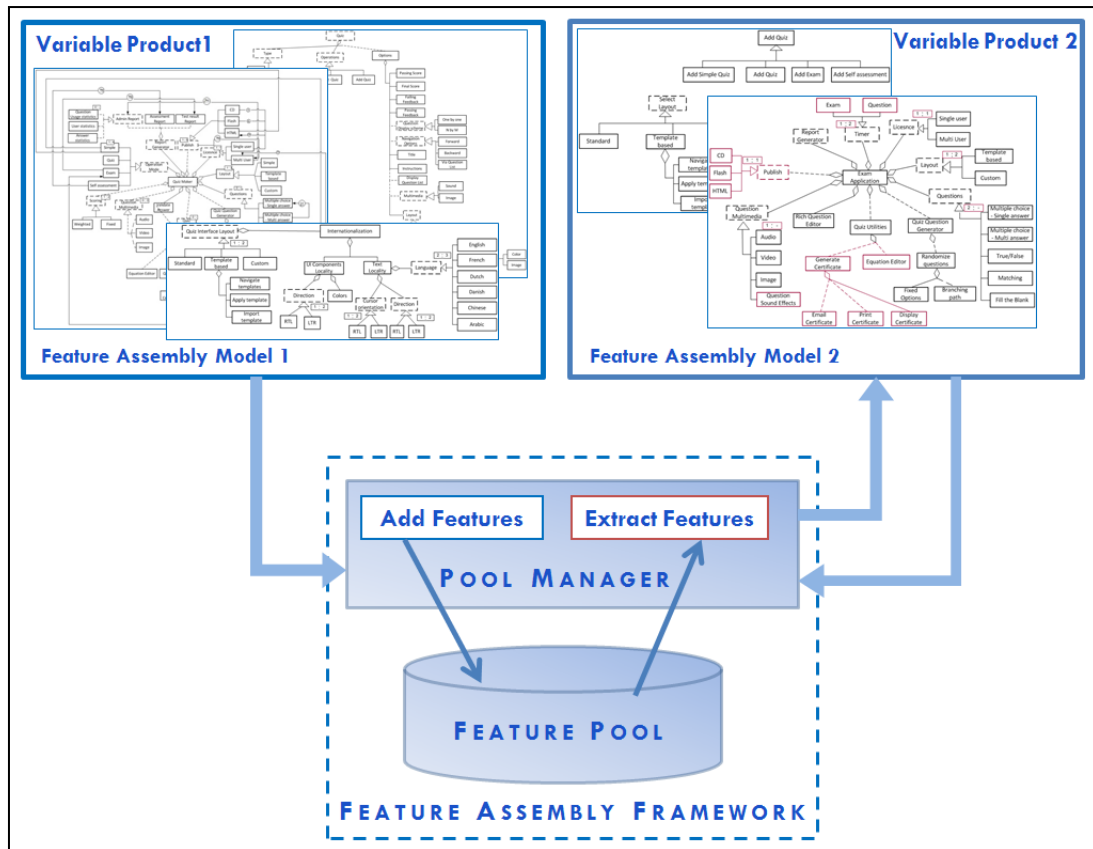


Figure 9.2: Feature Assembly Reuse Framework Overview

9.3 The Feature Pool

The Feature Pool is a repository of features or rather feature specifications, allowing reusing features in more than one product (line). It acts as a continuously growing central storage for all the features of the software products developed within the organization. The feature pool may hold features with different granularity levels and belonging to different products and different perspectives. Features in the feature pool do not represent code but are rather abstract representations of software functionalities or characteristics (see chapter 6 for the definition of feature). Furthermore, the features are stored together with meta-data describing them and crucial for being able to identify later on potential features for reuse. These meta-data include keywords to characterize the feature, and an informal description of the feature. The features are stored together with their Feature Assembly model (part) that specifies their composition and their possible contribution to variability. To allow reusing features from Feature Assembly models, the Feature Assembly Modelling technique (FAM) already anticipated to an important reusability principle, i.e. the *modularity principle* stating that components to be reused should be cohesive and loosely coupled. As already mentioned in chapter 6, a feature is defined based on whether it represents a concrete capability provided by the product (line) (i.e. concrete feature) or as a specification of some abstract capability (i.e. abstract feature). How a feature contributes to the variability of a specific product (line) is not inextricably associated with the feature. This information is represented by the *cardinality* part of the Feature Assembly model in the case of abstract features, and represented by the

are not saved to the Feature Pool, because they change according to the context in which the feature is used, while others will always hold (e.g. domain constraints) therefore these should be part of the information stored with the features in the Feature Pool. In addition, the *Reason* property is also associated with the feature dependencies stored in the Feature Pool so that the rationale of the dependency is not lost. Similarly, the *Owner* is stored. It should be noted that the set of meta-data defined here can be extended and adapted depending on the needs of the company for which the feature pool is created.

In the Feature Pool, we also keep track of the perspectives in which the feature is defined or used. For a perspective, we maintain the following properties: *definition date*, (*Owner*) *Stakeholder*, *Description* and *Keywords*. Furthermore, each perspective is linked to the product (line) in which it was defined.

9.3.1 Feature Pool Example

To illustrate how the features are extracted and stored into the Feature Pool we show this process for an excerpt of the System perspective for the QPL (shown in figure 9.4).

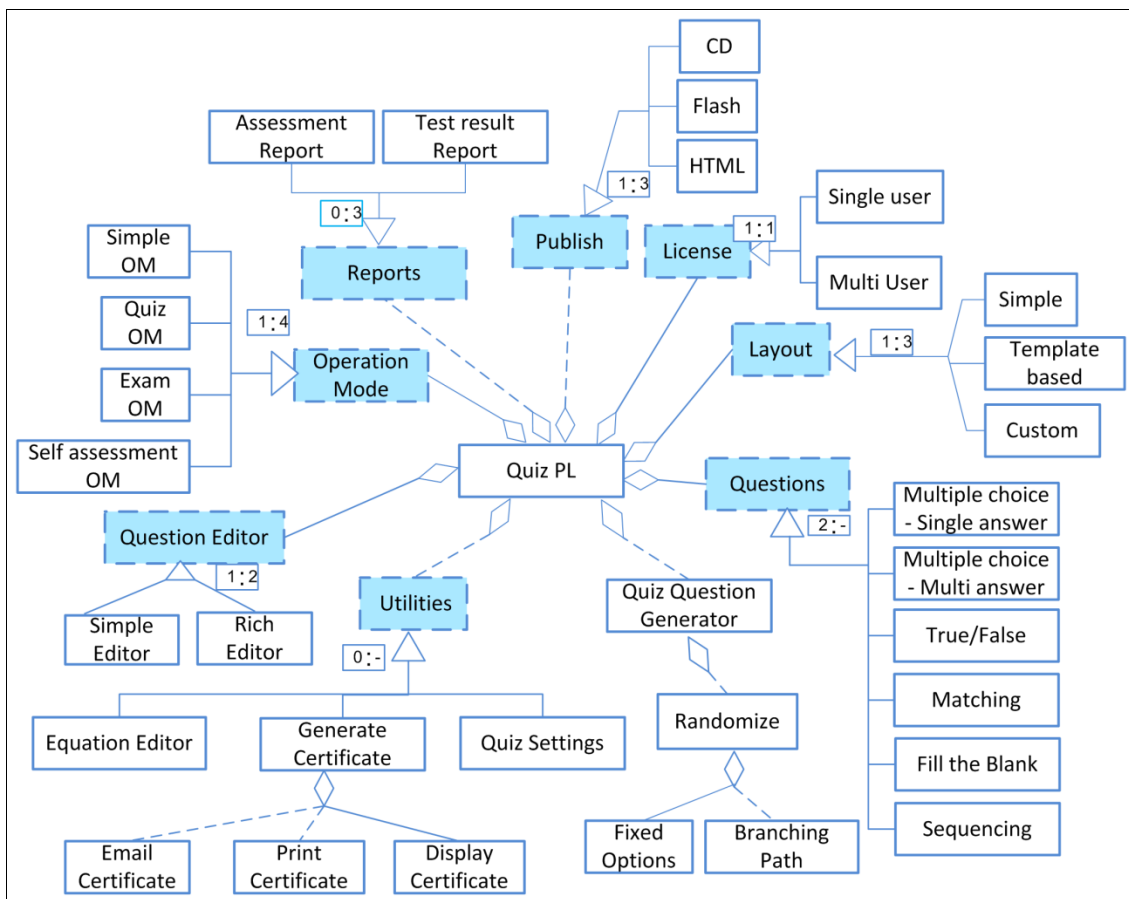


Figure 9.4: Excerpt of the System perspective Feature Assembly Model for the QPL

First we identify those features that are standalone; those represent candidate features to be added into the Feature Pool. Standalone features maybe concrete features that are added along with their compositions or abstract features that are added with their option features

(variants). Note that not all features will be added to the Feature Pool, nor all relations; features that are very specific to a certain application will not be added. In this example, such features are *Quiz PL*, which represents the quiz application; *Utilities*, which is a conceptual grouping of the utilities supported by the QPL; and *Quiz Question Generator*, which is very specific to the Quiz application.

Abstract features that will be added (with all their variants) to the Feature Pool are: *Publish*, *Layout*, *Questions*, *Reports*, *Operation Mode*, *License*, and *Question Editor*; these are all marked as *standalone* features (i.e. standalone property is set to ‘true’). This is shown in figure 9.5. Note that no cardinalities are added, as the cardinality is usually problem specific. Concrete features that will be added to the feature pool are: *Quiz Settings*, *Equation Editor*, *Generate Certificate* and *Randomize* which are all *standalone* features (i.e. standalone property is set to ‘true’). Note that *Generate Certificate* is a standalone feature composed of other non-standalone features (*Display Certificate*, *Print Certificate* and *Email Certificate*) and therefore they are also added to the feature pool under their parent feature (but with the stand-alone property set to ‘false’). Furthermore, while the *Utilities* feature itself was not added to the Feature Pool, the features *Equation Editor* and *Quiz Settings* were added to the Feature Pool, this is because these two features represent standalone functionality that can be reused on its own, independent of the application. The corresponding extracted Feature Pool is shown in figure 9.5; note that it represents clusters of features within the domain.

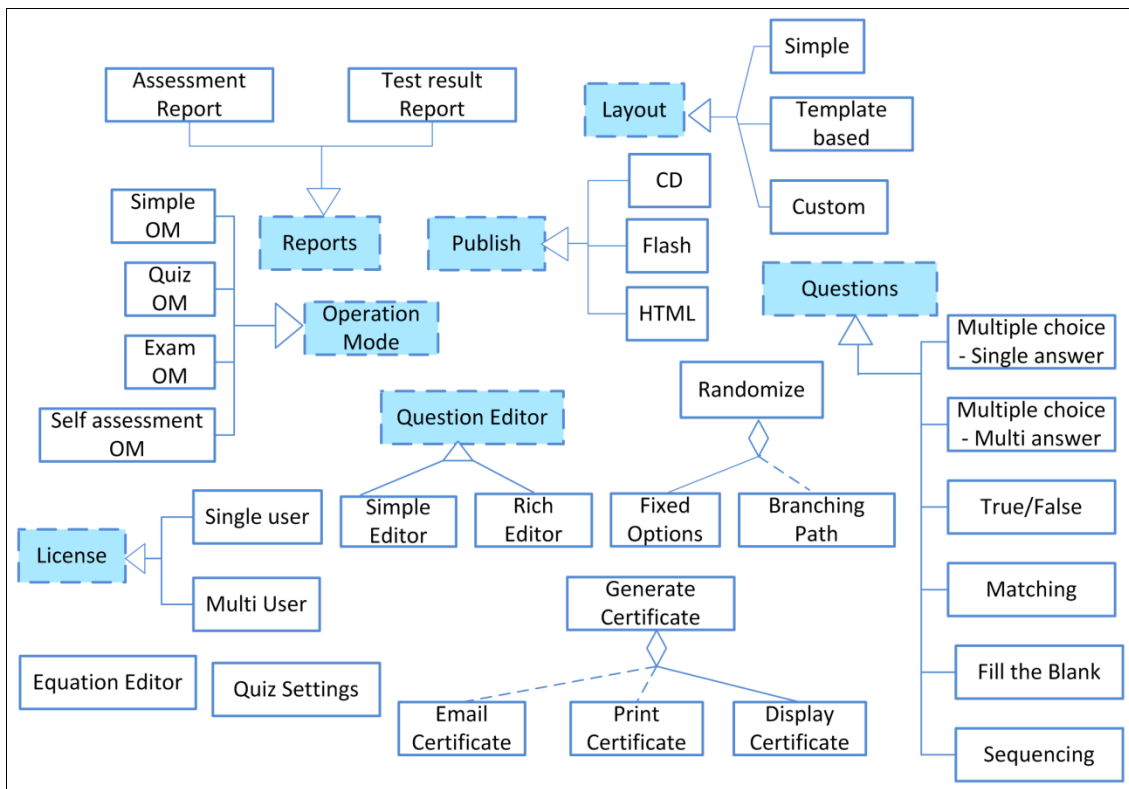


Figure 9.5: The Feature Pool Features Extracted from the QPL FAM of Figure 8.3

In terms of feature dependencies⁵⁰, the following dependencies are added (note the *Enforced* property associated with each feature dependency):

- Simple OM excludes Matching, Enforced: True

⁵⁰ Please refer to section 8.2.1 for the complete list of feature dependencies.

- Simple OM excludes Fill the Blank, Enforced: True
- Simple OM excludes Sequencing, Enforced: True
- Self Assessment OM excludes Single User, Enforced: False
- Exam OM excludes Single User, Enforced: False

Some feature dependencies are not added to the Feature Pool because they represent business/application constraints rather than domain constraints, such as: *Single User excludes CD*, and *Simple OM requires HTML*.

9.4 Assembling Features with Feature Assembly

As already mentioned the Feature Assembly Reuse Framework is a conceptual framework to create Feature Assembly models by reusing already existing features as well as new features. The process of the Feature Assembly approach is shown in figure 9.6, the following steps are identified:

1. **Analyse New Variable Product:** when a new product line (or even product) is required, the product (line) requirements are analysed to determine the new product (line) variability and commonality (as already mentioned in section 6.2).
2. **Identify Required Features:** the main features that characterize the product (line) should be identified. It should be investigated (e.g., via searching the Feature Pool) whether these features have been defined before (and therefore they already exist in the Feature Pool) or whether they are new and thus need to be further analysed. For existing features proceed to step 3, for new features proceed to step 4.
3. **Extract Existing Features from the Feature Pool:** Existing features are extracted from the Feature Pool (with their descendants). In addition, feature dependencies that govern these features should be extracted. The enforced properties should provide information on whether the dependency must hold or whether it is optional to include it.
4. **Define New Features:** new features are defined following the Feature Assembly Modeling technique, with the appropriate level of detail.
5. **Create Product Feature Assembly Models:** build up the Feature Assembly models that represent the required new product (line) combining both the new features and the existing ones.
6. **Add New Feature to Feature Pool:** populate the Feature Pool with the newly defined features along with their details as explained in the previous section

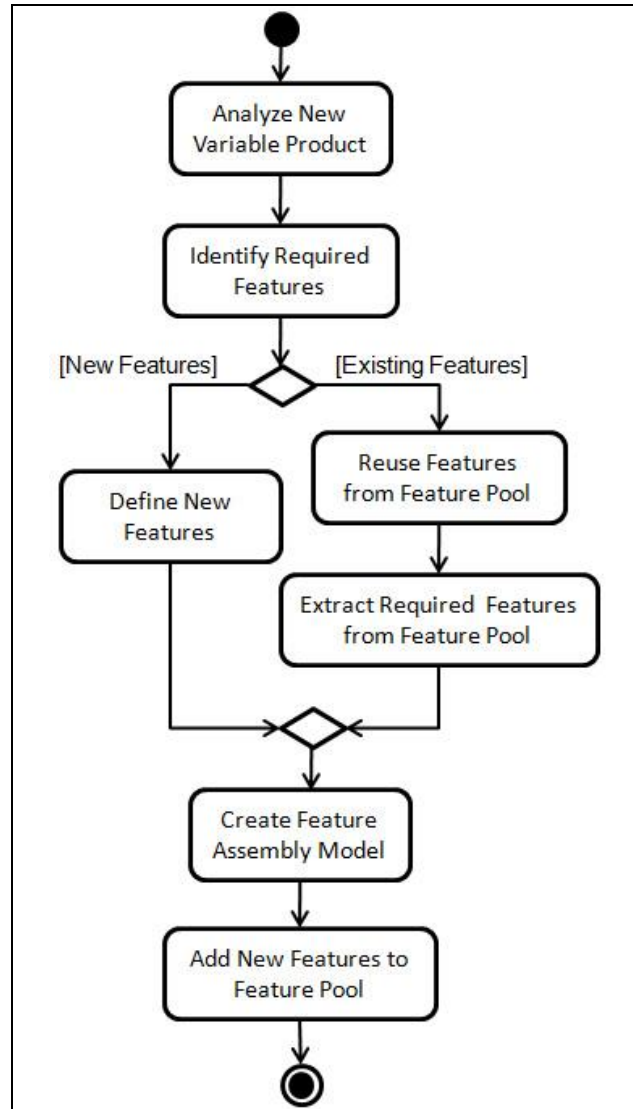


Figure 9.6: Feature Assembly Process

To guide the modeller on appropriate feature reuse possibilities we define the following set of reuse rules:

1. **Perspectives need to be respected:** when selecting features from the Feature Pool, the perspectives should be respected. This is because the semantics of the concept feature is different in different perspectives. For example, a *Spelling Check* feature in the Functional perspective indicates the functionality of checking the spelling. While a *Spelling Check* feature in the Graphical User Interface perspective indicates the visual appearance of the spelling checker. Therefore, the modeller should be careful when mixing features from different perspectives when reusing them; i.e. it is up to the modeller to maintain the semantics of the feature when reusing it.
2. **Reuse of abstract features:** abstract features may be extracted with *one or more* of their option features. Furthermore, new variants can be added to the set of allowed specifications (i.e. option features) of an abstract feature. When reused a cardinality (minimum and maximum) is attached to the feature.

3. **Reuse of option features:** an option feature in the Feature Pool (i.e. a variant feature) must be reused within the context of its parent (abstract) feature. An option feature is a specification of a certain (abstract) concept. Reusing the option feature together with the parent abstract feature helps maintaining this semantics. Furthermore, keeping this information allows to later on expand these abstract concepts with more specifications (either from the Feature Pool or with new ones) when needed.
4. **Reuse of concrete features:** when reusing concrete features their decomposition should be respected, i.e. all *mandatory decomposition relations* should be taken over, while optionally decomposition relations may be arbitrary transformed into mandatory relations or reused as they are or omitted.

We demonstrate these rules in the example below.

9.4.1 Feature Assembly Example

To illustrate how one can make use of *feature reuse* in Feature Assembly, consider the need for developing an *Exam* product line (Exam PL). The Exam product line is an application oriented for developing exams. Two types of Exams should be supported, simple exams and more advanced exams that contain advanced features such as multimedia associated with the questions. Two types of question editors should be supported, a simple editor, and an advanced editor.

Some features defined in the Quiz product line are applicable for reuse while others are only partially applicable. Using the Feature Assembly approach, the reusable features are looked-up and extracted from the Feature Pool. Features defined for the Quiz product line and applicable in the Exam PL include: *Reports*, *License*, *Questions*, *Randomize*, *Score*, *Operation Mode*, *Publish*, *Equation Editor* and *Layout*. This set of features includes some abstract features and some concrete ones. Figure 9.7 shows the Feature Assembly model for the System perspective of the Exam PL application; already existing features are shown in red text. Abstract features are extracted with *some* of their option features: the ones that are applicable for the Exam PL. For example, the *Layout* feature is extracted from the Feature Pool associated with two of its variant features *Template Based* and *Custom*. The *Layout* feature is also associated with a new cardinality based on the new situation, a minimum of one and a maximum of two, as shown in figure 9.7. Similarly, the abstract feature *Operation Mode* is reused associated with only two of its possible variants, the features *Simple OM* and *Exam OM*; a minimum of one and a maximum of two is specified. The feature *Reports* is reused with only two of its variants *Admin Reports* and *Test Result Report*, a minimum of one and a maximum of two is specified. The *Publish* feature is reused with only two option features associated to it, *Flash* and *HTML*. Furthermore, the *Question Editor* feature is reused as it is with two variants specified, *Simple Editor* and *Rich Editor*. Similarly, the *Score* feature is used as is.

Furthermore, new variants can be associated with already existing (i.e. in the Feature pool stored) abstract feature. In that case, these newly defined variants are added to the Feature Pool as new option features of that abstract feature. For example, the abstract feature *Question Multimedia* is reused with all its option features; in addition, a new option feature *Question Sound Effects* is also defined. The cardinality associated with the feature is a minimum of one and a maximum of four. Similarly, a new option feature *Essay* is added to set of option features associated with the *Questions* feature.

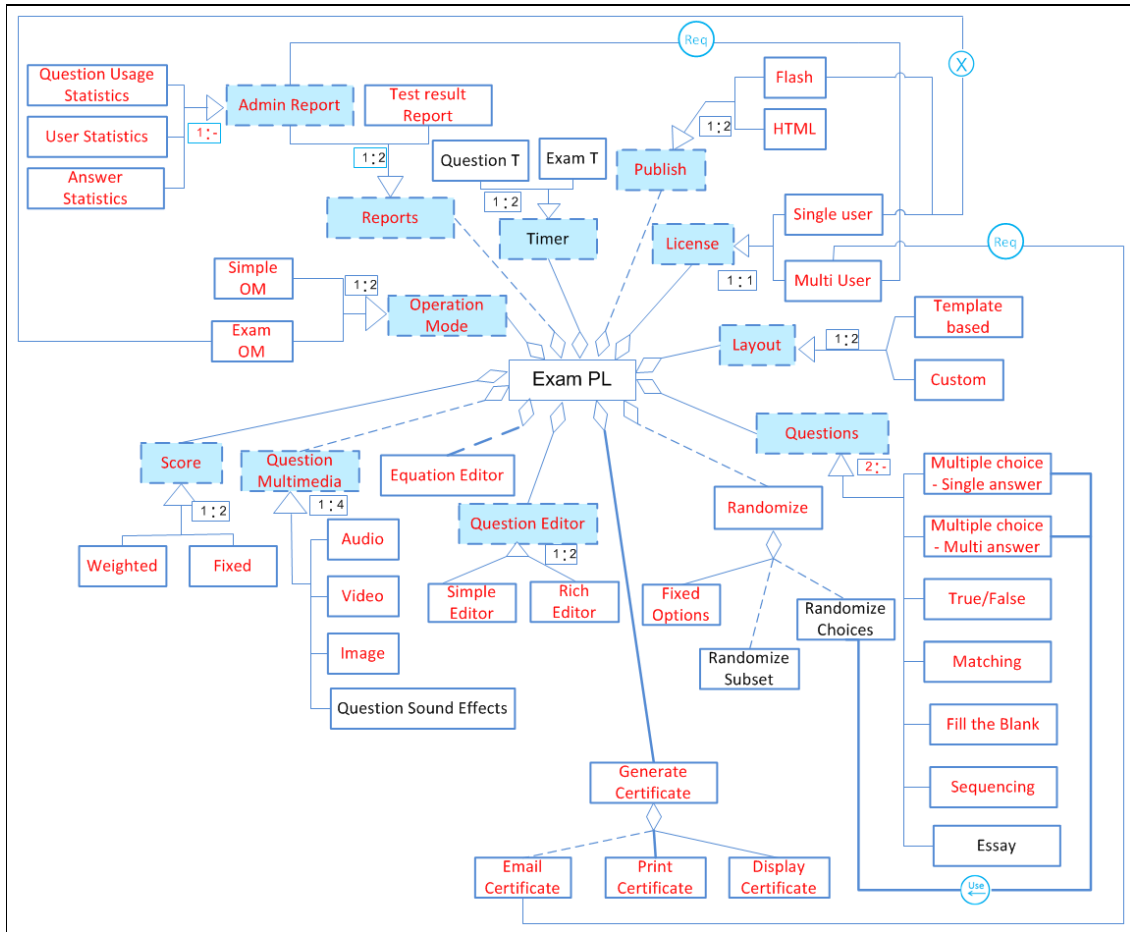


Figure 9.7: Feature Assembly Process

Concrete (standalone) features of the QPL are also reused (together with their decomposition defined for the QPL), because they are also applicable for the Exam PL. For example, the concrete feature *Equation Editor* is reused as an optional part of the Exam PL. Note that although *Equation Editor* was an option feature of the *Quiz Utilities* feature in the QPL, it was defined in the Feature Pool as standalone and therefore it could be reused without having to reuse its parent feature. The *Generate Certificate* feature (with all of its decompositions) is also reused, this time it is used as a mandatory part (as shown with a thick line in figure 9.7), while it was used as an optional feature in the QPL) of the Exam PL. Additionally, the feature *Randomize* is reused, with only its mandatory part, but two new optional sub-features of *Randomize* were added: *Randomize Subset*, which allows to randomize a subset of the questions, and *Randomize Choices* which allows to randomize the choices of the multiple choice questions.

In addition to the existing features, some new features were required for the Exam PL, such as the abstract feature *Timer*, which has the concrete option features *Question T*, and *Exam T* (which represent question timer and exam timer respectively).

It is clear in figure 9.7 that the number of red features, i.e. features reused from the Feature Pool exceeds the black features, i.e. newly defined features. It illustrates the flexibility of the Feature Assembly Modelling technique, which allows to reuse features while allowing them to contribute differently to the variability of a system. The newly added features should be added to the Feature Pool so that they are available to future developments.

9.5 Summary

In this chapter, we have presented our second major contribution of the thesis which is the Feature Assembly Reuse Framework that allows combining both variability and reusability at the design level by supporting the principle of feature modelling with reuse. The Feature Assembly Reuse Framework is a conceptual framework for creating feature models using already existing features as well as new features. The underlying principle was already adopted in industry and has led to establishing several parallel product lines reusing features from one another, thus increasing productivity. We adopt the same principle for defining variable software. We argue that the “feature” is an appropriate artefact to support this type of reuse.

Features are stored in a repository of features, referred to as the “Feature Pool”. The framework supports the population of the Feature Pool with existing features as well as newly defined ones, and this leads to the continuous growth of the Feature Pool. The Feature Assembly Reuse Framework supports the creation of Feature Assembly models by reusing already existing features and defining newly required features. The newly defined features are, in turn, added to the Feature Pool. Additionally, we have demonstrated the approach with an example. We have shown which features defined for the QPL are reusable and can be added to the Feature Pool. We have also demonstrated how these features could be reused for quickly defining the Feature Assembly models of a new product line, the Exam PL.

Chapter 10

Feature Assembly Knowledge Management Framework

In chapter 6, we have presented the Feature Assembly Modelling technique used to model software variability. We also presented the Feature Assembly Modelling language, which uses a graphical notation for modelling variable software. Feature Assembly models represent information concerning the features composing a variable product in addition to their variability and commonality. In addition, they hold information about feature interactions and dependencies. Therefore, they might help practitioners understanding and analysing the sources of complexity in their products as well as the sources of variability. When developing new products, various decisions need to be made concerning the features to be supported, the level of complexity supported, the coupling between features, the feasibility of the variability introduced, etc. Furthermore, variable software development is often faced with the challenge of change. Giving its complexity and often its large size, supporting the ability to change becomes a major concern. Change could be either changing existing features, adding new features, or refining existing ones. In these situations, Features Assembly models may be of great value as they allow tracing the impact of a change. For example, changing a feature may affect its composition, as well as its dependencies with other features. Therefore, it may lead to changes to other features that are dependent on the feature changed. In this chapter, we provide an answer for our third research question RQ2, *How can the knowledge in feature models and features be captured and unlocked?*

Feature Assembly models represent and document product information and in this way might establish a better communication between different stakeholders⁵¹. Nevertheless, although information is explicitly represented in the Feature Assembly models, they are still quite difficult to understand for non-modellers, and if models are stored as a whole it is not possible to query them for individual pieces of information and relate and combine information in different models. This becomes even more important for large products with many perspectives, many features, many feature-to-feature dependencies, and many stakeholders involved. This calls for a machine understandable representation of the Feature Assembly Models. Also, the consistency of the feature dependencies is difficult to check manually, with a machine understandable representation this could be automated. Therefore, firstly we need to answer the question RQ2.1, *how can the knowledge in feature models and features be captured?* Secondly, we need to answer the question RQ2.2, *how can communication and information sharing between the different stakeholders be supported in order to comprehend and find information concerning the features of the system, their dependencies, and variability?*

⁵¹ Stakeholders are users that need this information in the SPL's life cycle, both in domain engineering and application engineering (e.g. sales persons, product managers, marketing persons, developers, etc.) as discussed in section 5.2.

In this chapter, we answer these questions by providing a knowledge representation of Feature Assembly models. The purpose of this representation is to allow easy and flexible knowledge discovery in Feature Assembly models. Additionally, such a representation can act as a formal documentation that can be made available according to the needs of the different stakeholders during the product's life cycle. This knowledge representation is realized by mapping the Feature Assembly Modelling technique (provided in chapter 6) to an ontology that defines the concepts and semantics of the Feature Assembly Modelling technique. We call this ontology the *Feature Assembly Model Ontology* (FAM ontology). Firstly, we point out the merits of adopting an ontology-based knowledge representation technique for representing the information represented by Feature Assembly models, we present the FAM ontology and the rules that allow capturing Feature Assembly models (which answers our research question RQ2.1). Secondly, we provide the two possible scenarios for users to search for knowledge in the FAM Ontology, namely *knowledge browsing* and *knowledge querying*; we also show how the two can be combined in one dedicated browser that allows for knowledge querying and browsing while visualizing the information in the FAM notation (which answers our research question RQ2.2). We also provide some examples.

We also apply the same knowledge representation technique to define an ontology for the Feature Pool, called the *Feature Pool Ontology* (FP Ontology). The FP ontology is actually an excerpt of the FAM ontology as will be explained in section 10.3.

10.1 Overview

In order to provide a machine processable representation for our Feature Assembly models, we used the concept of ontology. This ontology, called the Feature Assembly Model Ontology or FAM Ontology, provides a conceptualization of the concepts and semantics of the Feature Assembly Modelling technique. In other words, the FAM Ontology provides a meta-model for Feature Assembly Models.

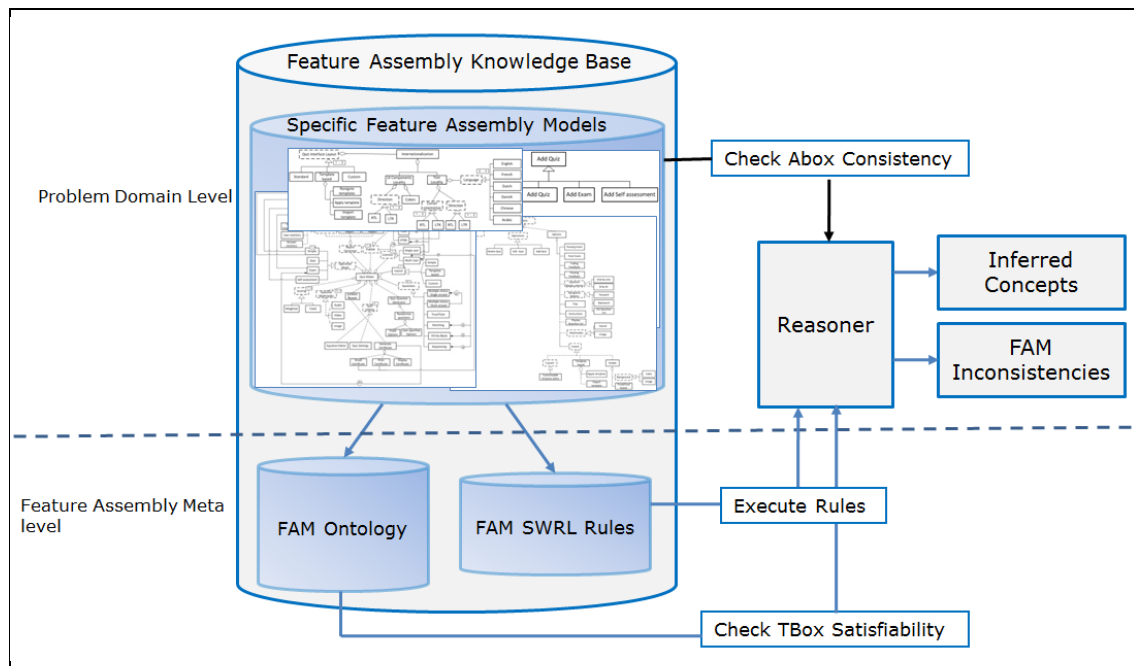


Figure 10.1: Overview of the Feature Assembly Knowledge Representation Framework

We have opted for an ontology, as software is available for ontologies to infer new knowledge from the existing knowledge. Furthermore, they allow reusing and extending the represented information. The reasoning support for ontologies can be used to check the consistency of the Feature Assembly models and to infer implicit information from the models. For this, we will use rules. Figure 10.1 shows the Knowledge Representation framework we defined for representing and manipulating Feature Assembly Models. At the meta-level both the *FAM Ontology* (TBox) (this will be covered in section 10.2.1) and the *FAM SWRL Rules* (this will be covered in section 10.2.2) are defined. The *FAM SWRL Rules* ensure the logical consistency of defined Feature Assembly models (i.e. the Abox). A Feature Assembly knowledge base exists when instances (i.e. individuals forming the Abox) of the ontology concepts are defined to represent specific Features Assembly models (as shown in figure 10.1) (this will be covered in section 10.2.3). A reasoner checks the consistency of the ontology and runs the SWRL rules against the ontology. The details of this framework will be explained in the next sections. Note that the Feature Assembly models are stored in one central repository and their relationships and overlapping concepts are represented explicitly (i.e. intra-perspective dependencies and features that belong to more than one perspective).

10.1.1 Why OWL?

We chose the Web Ontology Language (OWL) to represent our FAM Ontology. Particularly, we use the DL variant of OWL. OWL was chosen for the following reasons. Firstly, OWL ontologies are very popular these days for representing knowledge and many tools exist to support them, including tools for *browsing*, *visualizing*, *querying* and *editing* the ontology. Furthermore, activities such as knowledge sharing and collaborative activities are supported in these OWL supporting tools. OWL is a standard language making ontologies created using OWL sharable (on the web).

Secondly, OWL has constructs that allow gluing together information (e.g., owl:sameas), this is particularly important for obtaining the overall picture of the different Feature Assembly Models defined in the different perspectives. In addition, OWL supports the Open World Assumption (OWA)⁵² making it possible to reason on incomplete information. This is particularly important for allowing the representation and reasoning of incomplete feature assembly models. Therefore, allowing to reason on models of individual perspectives, as well as on the perspectives integrated. This allows different teams to work independently or negotiating and iterating over their models during integration.

Thirdly, OWL (DL) was designed to support DL reasoning on top of the ontology model. This enables using DL reasoners to infer knowledge and using rules implemented in SWRL to deduce new knowledge and to validate the models (for example validating that no conflicting dependencies exist in the models). We selected the DL dialect of OWL because it is both sound and complete (we only use DL safe SWRL rules).

For the implementation of both the FAM Ontology and the FP Ontology we used the Protégé OWL⁵³ ontology editor. For browsing the FAM ontology we used the jOWL Ontology browser⁵⁴. We query the ontology using OWL2Query⁵⁵ (plugin for Protégé), a query engine for SPARQL-DL. The reasoning is done using the Pellet⁵⁶ reasoner.

⁵² In OWA the lack of a given assertion or fact being available does not imply whether that possible assertion is true or false, it simply is *not known*. In other words, lack of knowledge does not imply falsity.

⁵³ Protégé OWL: <http://protege.stanford.edu/overview/protege-owl.html>

⁵⁴ jOWL Ontology Browser: <http://jowl.ontologyonline.org/jOWLBrowser.html>

10.2 The FAM Ontology

The FAM Ontology acts as a DL logic conceptualization of the Feature Assembly Model specification, i.e., the ontology defines the concepts and relationships defined for the Feature Assembly Modelling, namely: *features*, *feature relations*, *feature dependencies* and so on. In addition, the ontology also defines constructs that identify variability, i.e. *variation points* and *variants*. An OWL ontology expresses knowledge in terms of *classes*, *properties* and *restrictions*. Classes represent domain concepts or objects. Ontology development steps include the following phases based on the iterative engineering approach defined by Noy and McGuinness [2001]:

1. Define the concepts of the domain of discourse as classes in the ontology.
2. Arrange the classes in a taxonomic (subclass–superclass) hierarchy.
3. Define object properties (i.e. roles between different concepts) and describe allowed domains for these properties.
4. Define data properties (if any) and describe allowed domains (types) for these properties.

These steps represent actually a conceptualization of the domain of discourse. However, we have already done this conceptualization when defining our Feature Assembly Modeling approach. The result was the feature Assembly Meta Model (given in section 6.5 and shown again in figure 10.2), which defines the concepts in our domain of discourse. Conceptual data models and ontologies are quite similar, as both consist of conceptual relations and rules [Spyns, 2005]. Rules constrain how the concepts relate to each other, the restrictions that could hold on properties (called roles in ORM), and define inference guidelines. Several works show the relation between ORM conceptual modelling techniques and ontology engineering [Jarrar et al., 2003] [Spyns, 2005] [Keet, 2007], having identified the key concepts (in terms of ORM facts and roles) and how they relate, we base our mapping on these works.

We use the mapping defined by Keet [2007], to convert the conceptual model into an ontology:

1. Every non-lexical object type (solid ellipse in the conceptual meta-model of figure 10.2) is mapped into a concept in the ontology, i.e. an OWL class.
2. Subtypes in the conceptual meta-model are mapped to subclasses in the ontology to define the class hierarchies. If subtypes have an exclusion constraint in the conceptual model, it maps to a disjoint constraint between the subclasses in the ontology. Note that disjointness needs to be explicitly defined in OWL
3. Binary relations between two object types are mapped to properties between the two corresponding classes in the ontology.
4. Concepts defined by lexical object types (dashed line ellipse in the conceptual model of figure 10.2) where the values are not constrained, are mapped to data-type properties in the ontology if they represent free from data.
5. Concepts defined by lexical object types where the values are constrained to specific values are mapped to sub-properties the object type to which the lexical object type is connected.

⁵⁵ OWL2Query: <http://protegewiki.stanford.edu/wiki/OWL2Query>

⁵⁶ Pellet: <http://clarkparsia.com/pellet>

define a `Variability` class which has two subclasses `Variant` and `Variation Point`. Furthermore, as already mentioned in chapter 6, a feature is associated with a *binding time* and a *stakeholder* which can be either the owner of the feature or a user of this feature. We therefore define these concepts in the ontology by two classes, `Binding Time` and `Stakeholder` respectively. The FAM Ontology also defines a *Priority* concept, represented by the `Priority` class, to define how important the feature is. This is added to support decision making at later stages.

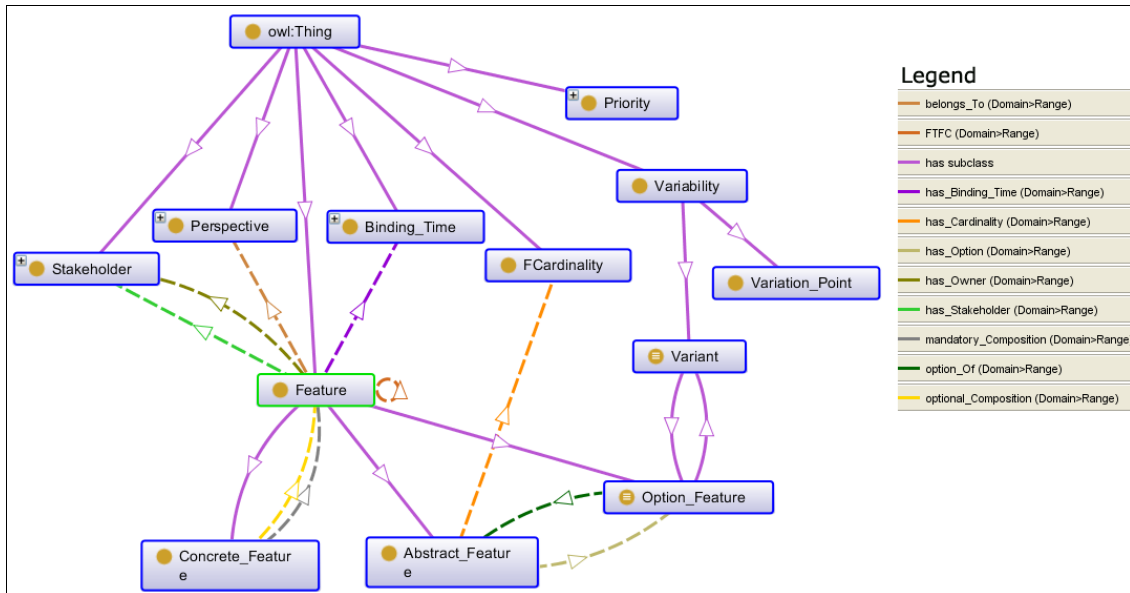


Figure 10.3: Corresponding FAM Ontology Meta-Model visualized by OntoGraf

10.2.1 The FAM Ontology Vocabulary

In this section, we provide a more detailed discussion on the structure of the FAM Ontology⁵⁷. As already mentioned, the ontology is represented by a set of classes that represent the Feature Assembly Modelling concepts. Additionally, the FAM Ontology also holds a set of properties that map the relations between the different classes and therefore establish a link between the concepts of the ontology (as shown in figure 10.3). These properties represent the feature relations and dependencies defined in the Feature Assembly Modelling technique. Within the FAM Ontology, each property is restricted to connect two specific classes. This restriction is defined by the *domain* and *range* characteristics of OWL properties. The domain holds a class description of the class(s) the property belongs to. The range holds a class description of the allowed classes this property can refer to (i.e. link to).

For each FAM concept defined in the FAM Ontology, we describe its class restrictions, properties, and property restrictions. We also provide how it is formally defined in OWL by means of Description Logic (DL) syntax⁵⁸.

⁵⁷ Appendix B shows the complete FAM Ontology represented in OWL Functional syntax.

⁵⁸ In this section we use DL to represent the formal semantics of the FAM ontology's TBOX in addition to the equivalence in OWL whenever a logic formula is used. This is because the DL syntax is more compact to use and more common in ontology modelling. Every OWL DL axiom has an equivalent DL representation, for more information about this representation please refer to OWL Web Ontology

1. Feature:

The class `Feature` has two object properties that relate to the stakeholders: `has_Stakeholder` and `has_Owner`, both object properties have domain `Feature` and range `Stakeholder` (as represented by axioms 1 and 2). The `has_Stakeholder` object property refers to the stakeholders involved with the features; we represent this with an `allValuesFrom`⁵⁹ restriction on the `has_Stakeholder` property (as represented by axiom 3). The `has_Owner` object property refers to the owner of the feature; each feature can only have at most one owner. To denote this we define a cardinality restriction of maximum one on the `has_Owner` object property (as represented by axiom 4). These facts are defined as follows:

$\exists has_Stakeholder.T \sqsubseteq Feature$	$T \sqsubseteq \forall has_Stakeholder.Stakeholder$... (1)
$\exists has_Owner.T \sqsubseteq Feature$	$T \sqsubseteq \forall has_Owner.Stakeholder$... (2)
$Feature \sqsubseteq \forall has_Stakeholder.Stakeholder$... (3)
$Feature \sqsubseteq \leq 1 has_Owner.Stakeholder$... (4)
$Feature \sqsubseteq \leq 1 has_Binding_Time.Binding_Time$... (5)
$Feature \sqsubseteq \leq 1 has_Priority.Priority$... (6)

Listing 10.1: DL Axioms Representing the Feature Class

The `Feature` class is also associated with a description via the `has_Description` data property, which has domain `Feature` and range the `String` data type. Furthermore, the `has_Binding_Time` object property identifies the binding time of a certain feature. The `has_Binding_Time` time property has domain `Feature` and range `Binding_Time`. Furthermore, a feature may not be associated with more than one binding time, this restriction is maintained by defining the `has_Binding_Time` property as functional, i.e. as an `owl:FunctionalProperty` (represented by axiom 5). The `has_Priority` object property associates a priority to a feature. Similar to the binding time, a feature cannot be associated with more than one priority, therefore, the `has_Priority` property is defined as an `owl:FunctionalProperty` (represented by axiom 6).

The set of feature dependencies defined in FAM (i.e. feature-to-feature constraints) are applicable to any type of feature and therefore they are defined as part of the `Feature` class properties. We define an upper level object property `FTFC` (Feature To Feature Constraint) which groups all the feature dependencies. The object property `FTFC` and all its sub-properties have domain `Feature`, and have range `Feature` (represented by axiom 7). The `FTFC` sub-properties are: `excludes`, `requires`, and `uses`; they are defined in OWL FAM Ontology as follows (as shown by axioms 8 to 10):

- `requires`: is defined as an `owl:TransitiveProperty` to represent the transitivity of this property.

Language Semantics and Abstract Syntax [<http://www.w3.org/TR/owl-semantics/>]. Appendix C shows the OWL DL to Description Logic semantics mapping.

⁵⁹ `allValuesFrom` Universal Restriction describes classes of individuals that for a given property only have relationships along this property to individuals that are members of a specified class. `someValuesFrom` Existential Restriction describes classes of individuals that participate in at least one relationship along a specified property to individuals that are members of a specified class.

- `excludes`: is defined as an `owl:SymmetricProperty` to represent the symmetry of this property.
- `uses`: is defined as an `owl:TransitiveProperty` to represent the transitivity of this property.

$\exists FTFC.T \sqsubseteq Feature$	$(T \sqsubseteq \forall FTFC.Feature)$... (7)
$requires \sqsubseteq FTFC$	$Transitive(requires)$... (8)
$excludes \sqsubseteq FTFC$	$excludes \equiv excludes^-$... (9)
$uses \sqsubseteq FTFC$	$Transitive(uses)$... (10)

Listing 10.2: DL Axioms Representing the Feature Dependencies

Furthermore, the *same* feature dependency of FAM, is represented by the `owl:sameAs` owl property, which allows two or more instances to be treated by the reasoner as the same individual. The `owl:sameAs` property is symmetric by default.

In order to add the feature dependency *Reason* property we have extended the OWL annotations with the following annotations: *Dependency_Reason* and *Dependency_Owner*. This allows us to define textual rationale to the dependency assertion axioms (more on this in section 10.2.3).

A feature is linked to a certain perspective via the `belongs_To` object property. `belongs_To` has domain `Feature` and range `Perspective` (as represented by axiom 11). A feature may belong to more than one perspective.

$\exists belongs_To.T \sqsubseteq Feature$	$T \sqsubseteq \forall belongs_To.Perspective$... (11)
---	---	----------

2. Abstract_Feature

The class `Abstract_Feature` is a subclass of `Feature` (as represented by axiom 12). It has a `has_Option` object property, which links the abstract feature to its corresponding option features. The `has_Option` property has domain `Abstract_Feature` and has range `Option_Feature` (as represented by axiom 13). Furthermore, an `allValuesFrom` restriction on the `has_Option` property restricts it to only values of type `Option_Feature` (as represented by axiom 14). An abstract feature is associated with a cardinality via the `has_Cardinality` property. An abstract feature cannot be associated with more than one cardinality, therefore, the `has_Cardinality` property is defined as an `owl:FunctionalProperty` (as represented by axiom 15).

$Abstract_Feature \sqsubseteq Feature$... (12)
$\exists has_Option.T \sqsubseteq Abstract_Feature$	$T \sqsubseteq \forall has_Option.Option_Feature$... (13)
$Abstract_Feature \sqsubseteq \forall has_Option.Option_Feature$... (14)
$Abstract_Feature \sqsubseteq \leq 1 has_Cardinality.FCardinality$... (15)

Listing 10.3: DL Axioms Representing the Abstract Feature Class

3. Concrete_Feature

The class `Concrete_Feature` is a subclass of `Feature` (as represented by axiom 16). It has a `Composition` object property, which groups the different types of compositions allowed by a concrete feature (the mandatory composition and optional composition). Therefore, the `composition` object property has two sub-properties namely: `mandatory_Composition` and `optional_Composition` (as represented by axiom 17). A concrete feature is allowed to be composed of concrete features and abstract features which are not option features (this is represented by axioms 18 and 19).

$Concrete_Feature \sqsubseteq Feature$... (16)
$composition \sqsubseteq mandatory_Composition \sqcup optional_Composition$... (17)
$Concrete_Feature \sqsubseteq \forall mandatory_Composition. (Concrete_Feature \sqcup Abstract_Feature \sqcup \neg Option_Feature)$... (18)
$Concrete_Feature \sqsubseteq \forall optional_Composition. (Concrete_Feature \sqcup Abstract_Feature \sqcup \neg Option_Feature)$... (19)

Listing 10.4: DL Axioms Representing the Concrete Feature Class

4. Option_Feature

The class `Option_Feature` is a subclass of `Feature` (as shown by axiom 20). It has an `option_of` object property which refers to its parent abstract feature. The `option_of` object property is the inverse property of the `has_option` object property of the `Abstract_Feature` class (as shown by axiom 21); this property is defined to facilitate the querying. As already mentioned (please refer to chapter 6), option features indicate choices or alternatives (based on the associated cardinality), therefore they represent variants of a certain variation point (i.e. of their parent abstract features). We represent this by defining `Option_Feature` as an equivalent class for the `Variant` class (as shown by axiom 22).

$Option_Feature \sqsubseteq Feature$... (20)
$Option_Feature \sqsubseteq \forall option_Of. Abstract_Feature$... (21)
$Option_Feature \equiv Variant$... (22)

Listing 10.5: DL Axioms Representing the Option Feature Class

5. FCardinality

The class `FCardinality` defines the feature cardinality associated with abstract features. As already mentioned, each abstract feature should be associated with a cardinality that states the minimum and maximum number of features allowed in a valid configuration. To map this into the ontology, we have defined two data properties `max` and `min` associated with the `FCardinality` class (i.e. have domain `FCardinality`); `max` represents the maximum cardinality and `min` represents the minimum cardinality; both are define as literals (as represented by axiom 23).

$\exists \text{min.T} \sqsubseteq \text{FCardinality}$	$\exists \text{max.T} \sqsubseteq \text{FCardinality}$... (23)
--	--	----------

6. Perspective

The class `Perspective` is a high level class that groups all possible perspectives. A specific perspective is an instance of this class (or any of its sub-classes). The FAM Ontology defines the following types of perspectives as subclasses of the `Perspective` class: `System`, `User`, `Functional`, `Graphical_User_Interface`, `Task`, `Hardware_Interface`, `Localization`, `Non_Functional`, and `Persistent`.

7. Stakeholder:

The class `Stakeholder` is a high level class that groups all possible stakeholders. A specific stakeholder is an instance of this class. The FAM Ontology defines the following type of stakeholders as subclasses of the `Stakeholder` class: `Marketing`, `Modeller`, `Sales`, `Testers`, `Business_Analyst`, `Client`, `Developer`, and `Domain_Expert`. The FAM Ontology provides this classification for stakeholders, but if a different classification exists in another upper ontology or a different classification is more appropriate it can be changed. Actual stakeholders are defined as instances of these classes.

8. Binding_Time

The class `Binding_Time` represents the time of binding for the variability options of a feature. A set of all the different possible binding times are defined as individuals (i.e. instances) of the binding time class. The FAM Ontology defines the following binding times: `Analysis`, `Design`, `Compilation`, `Implementation`, `Installation`, `RunTime`, and `StartUp`.

9. Priority

The class `Priority` defines, as instances, all different possible priorities a feature may have. The FAM Ontology defines the following instances of the `Priority` class: `Severe`, `Top`, `High`, `Medium`, `Low`, and `None`.

10. Variability

The class `variability` is a high level class that groups the two types of variable features namely variation points and variants. They are represented by two sub-classes `Variant` and `Variation_Point`. As already mentioned, the class `Variant` is equivalent to the class `Option_Feature`.

There are two cases to identify a feature as a variation point. In the first case, it concerns an abstract feature that has some variants (i.e. option features). The second case is when a concrete feature is part of an optional composition. To assign features satisfying these conditions to the `Variation_Point` class, we define two SWRL rules that capture these two cases. These rules are shown by axioms 24 and 25. Axiom 24 states that if x is a concrete feature, and it has an optional composition y then x should be a variation point. Axiom 25 states that if x is an abstract feature, and it has at least

one option feature y then x should be a variation point. Note that abstract features with no option features assigned are not (yet) treated as a variation point.

$$\text{Concrete_Feature}(?x) \wedge \text{optional_Composition}(?x, ?y) \rightarrow \text{Variation_Point}(?x) \quad \dots (24)$$

$$\text{Abstract_Feature}(?x) \wedge \text{has_Option}(?x, ?y) \rightarrow \text{Variation_Point}(?x) \quad \dots (25)$$

Listing 10.6: DL Axioms Representing the Rules that derive the Variation Points and Variants

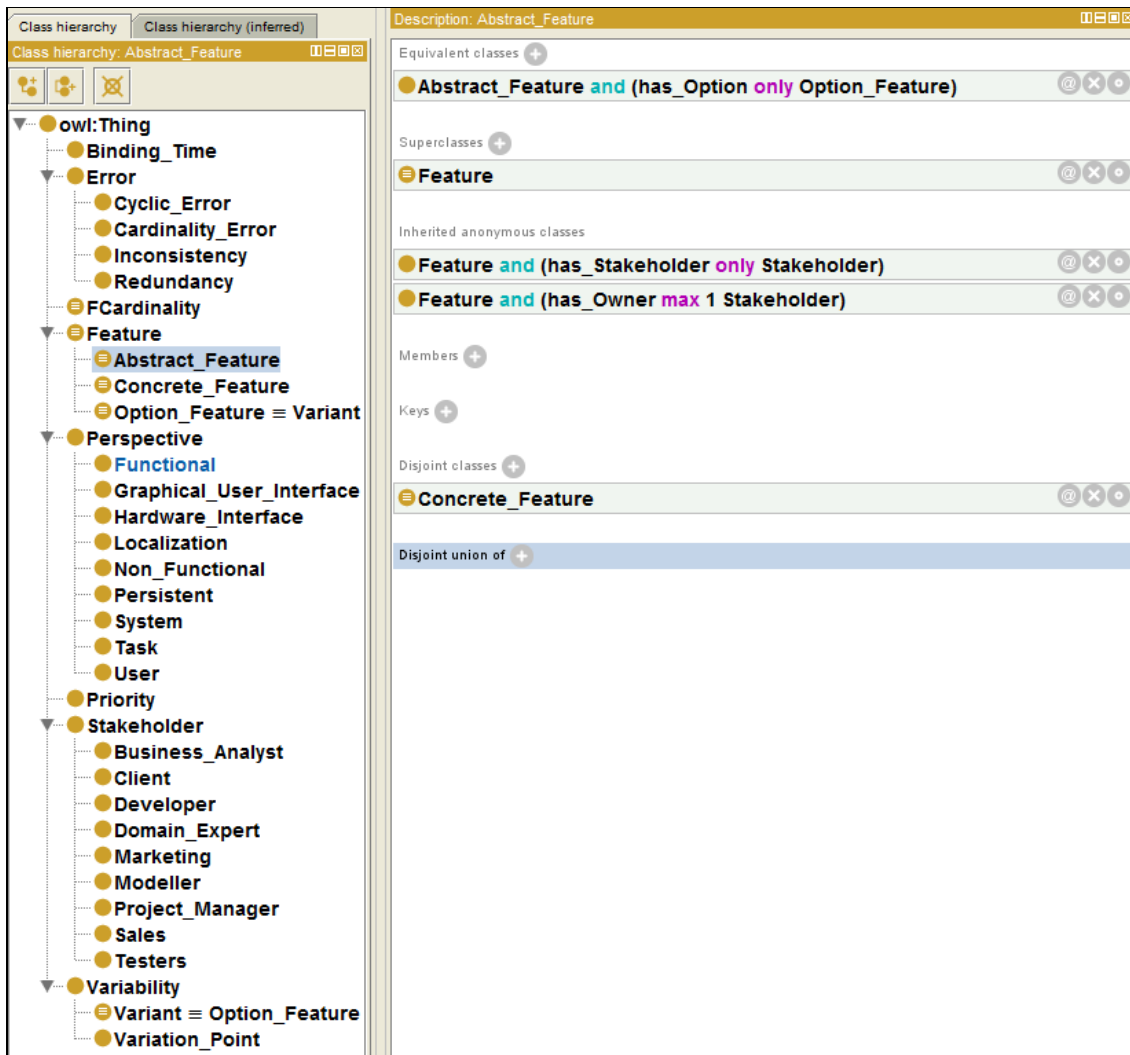


Figure 10.4: FAM Ontology Class Hierarchy Shown in Protégé

Furthermore, in addition to the above-mentioned semantics, some additional *Class Disjoint* restrictions are required because of the open world assumption used by OWL. These are added to provide guidance to the reasoner while inferring new deductions. We use the OWL disjoint class restriction `owl:disjointWith` to explicitly state the disjoint classes and therefore indicate that individuals (i.e. concepts in FAM) cannot belong to more than one class from the disjoint group. For example, `Feature` is disjoint with the following classes: `Perspective`, `FCardinality`, `Binding_Time`, `Stakeholder`, and `Priority`. Similarly, `Perspective` is disjoint with `Feature`, `FCardinality`, `Binding_Time`, `Stakeholder`, `Variability` and `Priority`.

Figure 10.4 shows a screenshot from the FAM Ontology in Protégé showing the defined class hierarchy.

10.2.2 FAM Error Detection via the FAM Ontology

It is important to support the modeller as much as possible in creating feature assembly models. Automatic error detection should be part of this. One of the benefits of using owl for the FAM Ontology is owl's inference capabilities, which are usable for identifying possible modelling flaws. Modelling flaws should be highlighted to the modeller in order for him/her to take the appropriate decisions for correcting them, as models with modelling flaws may lead to invalid configurations.. It is important to support the modeller in detecting and understanding sources of errors. As the size of the software grows, it becomes more and more difficult to spot errors manually. Furthermore, using perspectives eases the modelling process of the individual feature assembly models, but at the same time, it increases the complexity of the consistency checking process. This is because the consistency of the overall model is determined by the consistency of each perspective in addition to the consistency of the overall model (as already mentioned in chapter 6).

An important advantage of our FAM ontology is that it allows integrating the Feature Assembly models constructed for different perspectives into one model with no cost. That is because a perspective is represented as a property of a feature in the FAM Ontology. Furthermore, perspectives are related via intra-perspective dependencies. This is extremely useful in the case of very large system and/or systems with many feature dependencies, as it allows gluing together the different Feature Assembly models created in the different perspectives. Furthermore, in the ontology there is no real distinction between inter-perspective dependencies and intra-perspective dependencies, the only difference is whether the two features involved in the dependencies coexist in the same perspective or not. This allows considering feature dependencies irrelevant of their type (i.e. inter-perspective or intra-perspective); therefore no extra computation is required.

Before elaborating on how we support error detection, let us consider a simple example of an error. Consider the following set of feature dependencies:

<i>FeatureA requires FeatureB</i>	(1)
<i>FeatureB uses FeatureC</i>	(2)
<i>FeatureC excludes FeatureA</i>	(3)

Listing 10.7: An Example Showing Possible Modelling Errors

Bearing in mind that *requires* and *uses* are transitive dependencies; dependencies (1) and (2) yield that FeatureA, FeatureB, and FeatureC could exist in a valid product configuration. While dependency (3) yields that FeatureA and FeatureC cannot co-exist together. Therefore these sets of dependencies are contradictory, and actually lead to a semantic inconsistency in the Feature Assembly models represented by the FAM Ontology individuals, i.e. the ABox. It should be noted that the perspectives to which the features belong is in this case not relevant. We call contradictory feature dependencies an inconsistency.

In order to support error detection, we have identified a number of common types of errors that may occur in Feature Assembly models, and for which the FAM Ontology will provide support:

- **Cyclic Dependencies:** this type of logical inconsistency results from cycles in the Feature Assembly model. From a modelling point of view, these cycles imply that

the features are not well defined (i.e. hold too much coupling) or are over-decomposed. The cycles result from the feature dependencies that are asymmetric properties (i.e. *uses*). For example, listing 10.8 shows a *uses* chain in axioms 1 to 3, the chain is then closed by axiom 4 which results in *FeatureA uses FeatureA*.

<i>FeatureA uses FeatureB</i>	(1)
<i>FeatureB uses FeatureC</i>	(2)
<i>FeatureC uses FeatureD</i>	(3)
<i>FeatureD uses FeatureA</i>	(4)

Listing 10.8: An Example Showing a Cyclic Error

- **Inconsistent Dependencies:** these errors result from a logical contradictory or from conflicting feature dependencies. This conflict will result in a wrong product configuration or even prohibit finding a configuration at all. This is due to using mutually exclusive feature dependencies, as already mentioned in chapter 6 this list is: (*requires, excludes*), and (*uses, excludes*). The example given above (listing 10.7) is an example of an inconsistency.
- **Redundant Dependencies:** These “errors” result from using dependencies that imply the same as some other feature dependencies specified. The larger the model and the more people involved in the modelling, the more redundancy will be introduced (usually accidentally). Redundancy is not a real error but is in general considered as bad modelling practise and should at least be pointed out. Listing 9.9 shows an example, axioms 1 and 2 imply that *FeatureA uses FeatureC*, this makes axiom 3 redundant with the conclusion provided by axioms 1 and 2.

<i>FeatureA requires FeatureB</i>	(1)
<i>FeatureB uses FeatureC</i>	(2)
<i>FeatureA requires FeatureC</i>	(3)

Listing 10.9: An Example Showing Redundant dependency

- **Cardinality Errors:** For a cardinality, the minimum should not exceed the maximum, if this is not the case then a cardinality error holds.

Note that the FAM ontology will not check for *dead features*. A feature is dead if it cannot appear in any of the products of the software product line, for instance when a mandatory feature excludes an optional feature, then this optional feature can never be selected (i.e. is a dead feature). Therefore, in order to detect dead features, one must solve the constraint problem represented by the Feature Assembly model; the feature(s) that do not appear in any valid solution would then be marked as a dead feature(s) [Benavides et al., 2010]. The FAM Ontology does not have the objective of finding solutions for this constraint problem; it is rather a representation of it.

Although the above-mentioned situations could have been forbidden by the feature assembly language definition, in practice this would result in a very stringent modelling

process, which is not desirable. Modelling is an iterative process and so-called errors occurring in one iteration may be left on purpose because they will be taken into consideration in a next iteration. Therefore, we prefer not to check (or forbid) errors and flaws during the modelling itself (e.g., by the modelling editor) but have it as a separate process. Using the FAM ontology for detecting errors also allows providing the source of the error (see section 10.2.2.2) and thus eases the finding of a solution.

In order to deal with errors we have defined a class `Error` in the FAM Ontology that allow capturing the four different types of modelling errors mentioned. The `Error` class has four subclasses that refer to the different error types captured, namely: `Cyclic_Error`, `Inconsistency`, `Redundancy`, and `Cardinality_Error`. The class `Cyclic_Error` will hold all the features that contain in their specifications a cycle between two features. We have defined an object property `cyclic`, which captures a cycle between two features. The object property `cyclic` is a symmetric property that has as domain and range the class `Cyclic_Error`. The class `Inconsistency` holds all the features that contain in their specifications something that leads to an inconsistency. Furthermore, an inconsistency usually occurs between two features (e.g., in the above example the inconsistency was between `FeatureA` and `FeatureC`); therefore we defined an object property `inconsistent`, which captures an inconsistency between two features. `Inconsistent` is a symmetric property that has as domain and range the class `Inconsistent`. `Redundancy` errors will be captured by the class `Redundancy`. Furthermore, the object property `redundant` captures redundancy between dependencies of two features; it has as domain and range `Redundancy`. To capture a cardinality error, we have defined the class `Cardinality_Error` which holds all the feature cardinalities (members of `FCardinality`) that contain such an error.

Next we define the rules that capture these errors in the FAM Ontology.

10.2.2.1 FAM Ontology - Error Capturing Rules

To detect Feature Assembly modelling errors, we have defined a set of SWRL rules. Furthermore, the defined SWRL rules also isolate the errors in the FAM Ontology by defining them as members of the appropriate `Error` class. We will explain below the different rules used.

- **Rules to Capture Cyclic Dependencies**

A cycle occurs when a feature dependency holds between *Feature A* and *Feature B*, and the same dependency holds between *Feature B* and *Feature A*. The following rule captures a cyclic use of the *uses* feature dependency. Note that the *uses* dependencies is transitive and therefore the cycle is not necessary a straightforward cycle but may be the result of a chain of feature dependencies (which was probably unforeseen at modelling time).

```
uses(?x,?y) ^ uses (?x,?y) → cyclic (?x,?y) ... (1)
```

Listing 10.10: Rule to capture Inconsistency Error due to cyclic *uses* dependency

- **Rules to Capture Inconsistent Dependencies**

An Inconsistency error is due to the use of two mutually exclusive dependencies. Usually, this error does not result from a direct specification of these mutually exclusive properties, but is a result of evaluating the existing feature dependencies and inferring new knowledge

based on their specifications. Rules (2) to (3) shown in listing 10.11, capture these type of errors, as already mentioned mutually exclusive feature dependencies are: (*requires*, *excludes*), and, (*uses*, *excludes*).

```
requires(?x,?y) ^ excludes(?x,?y) → inconsistent (?x,?y) ... (2)
uses(?x,?y) ^ excludes(?x,?y) → inconsistent (?x,?y) ... (3)
```

Listing 10.11: Rules to capture Inconsistency Errors due to conflicting dependencies

• Rules to Capture Redundant Dependencies

Redundancies may intentionally be part of the model or they may be an indication of badly defined feature dependencies. For example, *uses* and *requires* dependencies should not be used in combination, a feature may either *use* or *require* another feature but not both. Neglecting such cases may lead to implicit cycles in the defined Feature Assembly models. Furthermore, because some dependencies are defined as transitive, the reasoner will infer new feature dependencies based on this transitivity (i.e. it will compute the whole transitivity chain of the defined dependency). Rule 4 shown in listing 10.12, captures this redundancy between the (*uses*, *requires*) dependencies.

```
uses(?x,?y) ^ requires(?x,?y) → redundant (?x,?y) ... (4)
```

Listing 10.12: Rules to Capture Redundancy Errors Due to Redundant Dependencies

• Rules to Capture Cardinality Errors

Cardinality errors are due to human mistakes, i.e. by defining a maximum cardinality lower than the minimum. To capture such errors we use the SWRL built in `swrlb:greaterThan` to compare the maximum and minimum cardinalities (rule (5) shown in listing 10.13).

```
max(?x,?y) ^ min(?x,?z) ^ swrlb:greaterThan(?z, ?y) → Cardinality_Error(?x) ... (5)
```

Listing 10.13: Rules to Capture Cardinality Errors

Note that, the *Perspective* that the feature belongs to has no effect on the rules that identify feature-modelling errors. This is because the FAM Ontology glues the different perspectives based on common features (i.e. same features). Therefore, the above-defined rules are also applicable for features belonging to different perspectives.

10.2.2.2 FAM Ontology - Error Debugging

A merit of using the Protégé ontology editor is its capabilities to show the axiom entailments that lead to a certain inference, i.e. for each inference made by the reasoner an “*Explain Inference*” button exists, when pressed it provides the set of axioms that lead to the inference made. This is particularly important in the FAM Ontology in order to understand why certain premises were made by the reasoner when inferring knowledge. We also use this functionality to identify the axioms in the ontology (and therefore the model assumptions) that lead to errors. This is essential to support modeller in correcting errors. It is not sufficient to

report that errors are detected, we should also help the modellers as much as possible in correcting the errors and this implies that we should be able to show him the sources of the errors. We call this process the *error-debugging process*. To show how error-debugging works, we have added some *incorrect* feature dependencies to the QPL case given in chapter 8.

For example, to demonstrate inconsistency errors we added the following feature dependency to the already existing dependencies, *Simple OM requires Reports*. Figure 10.5 shows the entailed class assertions inferred by the reasoner. The entailment shows that *excludes* is a symmetric property, *requires* is a transitive property. It also states the rule: $\text{excludes}(?x, ?y) \wedge \text{requires}(?x, ?y) \rightarrow \text{inconsistent}(?x, ?y)$ on which the inferred information was based, and a list of the axioms that the rule(s) was evaluated against, in this case this list is:

Multi_User excludes Simple_OM	(1)
Reports requires Multi_User	(2)
Simple_OM requires Reports	(3)

Evaluating axioms (2) and (3) yields the inferred axiom: *Simple_OM Requires Multi_User*, this evaluated together with axiom (1), and the given SWRL rule results in the inferred axiom: *inconsistent (Multi_User, Simple_OM)*. Given that the *inconsistent* object property has a domain *Inconsistency*, then the reasoner assigns the individuals *Multi_User* and *Simple_OM*, as members of the *Inconsistency* class.

To demonstrate cardinality errors we added a wrong cardinality to the already existing feature *Publish_Cardinality*, which identifies the cardinality of the *Publish* feature. We set a Maximum cardinality of 1 and a minimum of 3. Figure 10.6 shows the explanation for the reasoner inference which associated the *Publish_Cardinality* feature to the class *Cardinality_Error*.

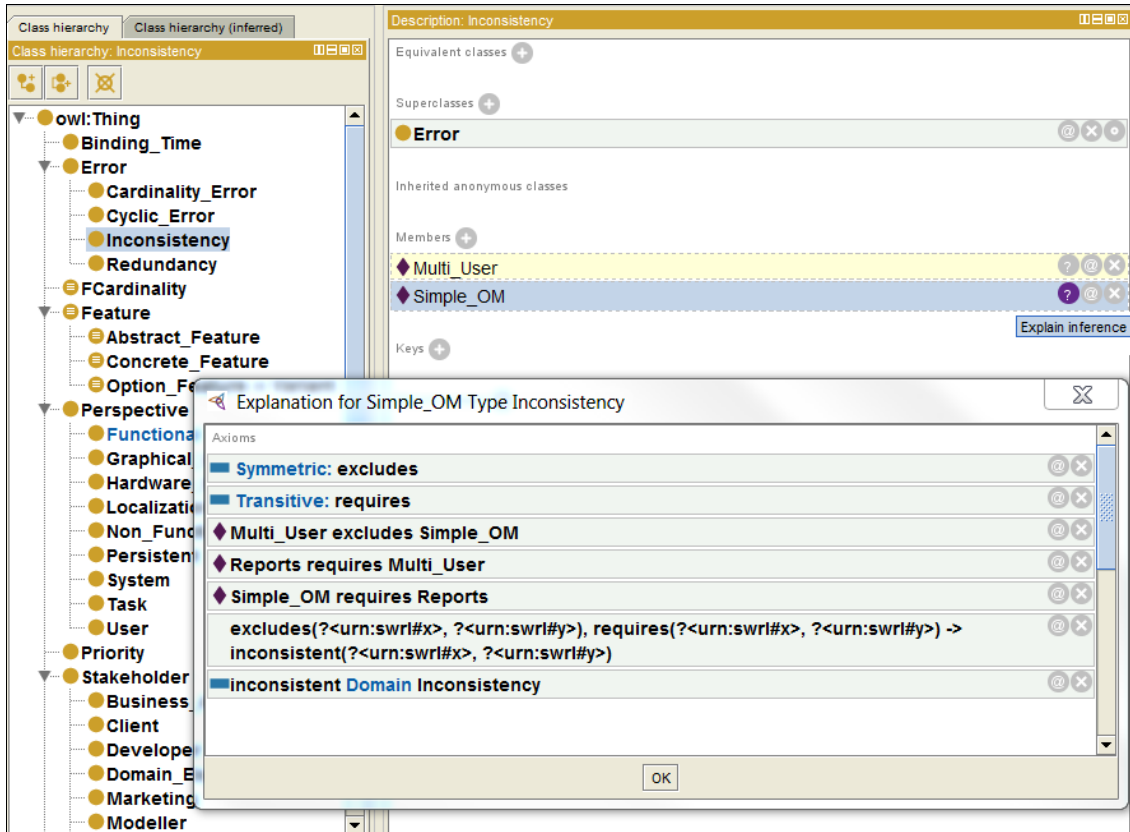


Figure 10.5: Explanation For an Inconsistency Detected by The Reasoner – Using Protégé

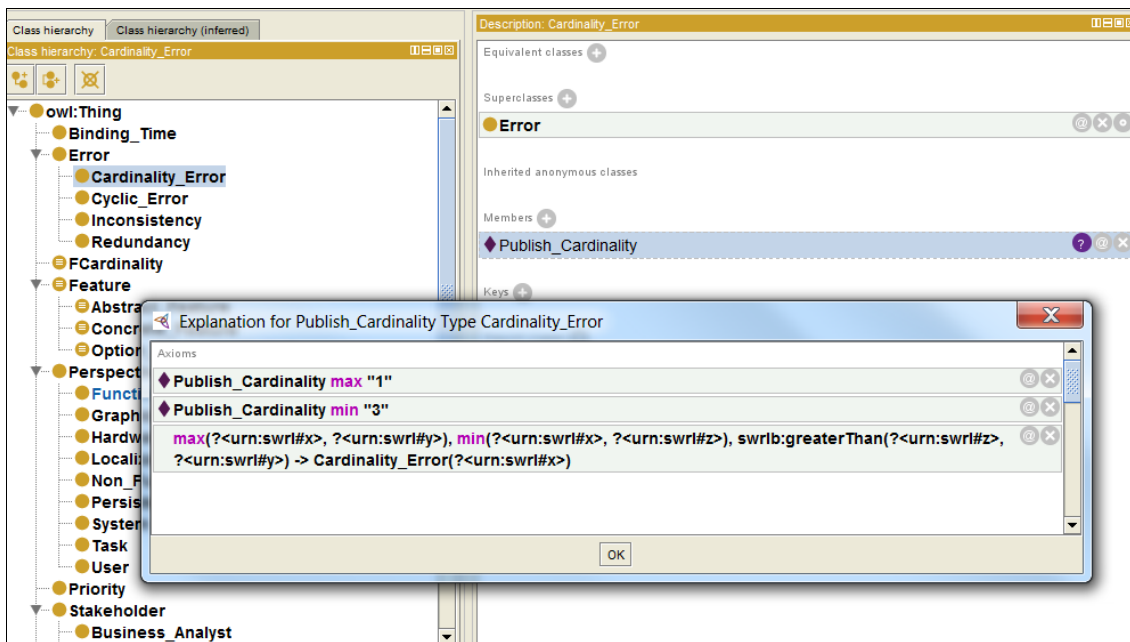


Figure 10.6: Explanation for a Cardinality Error- Using Protégé

It must be noted that the explanation for the reason of the error provided by protégé (as shown in figures 10.5 and 10.6) may not be very intuitive for modellers. This is due to the fact

that it was developed for debugging ontologies rather than debugging feature models. Therefore, it displays the raw axioms of the ontology that cause the inconsistency. In order to be usable for non-ontology specialists, an additional layer should be provided that translates the raw information provided into more meaningful FAM terminology. Nevertheless, it is a first and important step towards the facility to effectively debug errors in Feature Assembly models.

10.2.3 Populating the FAM Ontology with Individuals

As already mentioned, the FAM Ontology⁶⁰ acts as a meta-model to represent Feature Assembly models. The FAM Ontology benefits from the expressivity of OWL and the reasoning power of current OWL DL reasoners in order to infer hidden and implicit knowledge about the Feature Assembly models. This is helpful in detecting implicit or unintended modelling errors as shown in the previous section. The TBox of the FAM Ontology holds the actual representation of the specific Feature Assembly model(s). The ABox acts as the meta-model. As already mentioned, once the FAM Ontology is populated with individuals, it actually becomes a knowledge base containing knowledge on the Features Assembly models (please refer to figure 10.1)

In this section, we illustrate how the FAM Ontology can be populated. For this we use an excerpt⁶¹ (shown in figure 10.7) of the QPL case presented in chapter 8. We show the axioms that define the different individuals (i.e. feature assembly model instances) by means of the OWL functional syntax (this is done visually with an ontology editor). First we define the *Quiz_PL_System_Perspective* System perspective for the QPL, this is an instance of (i.e. class assertion) the *System* class (a subclass of the *Perspective* class):

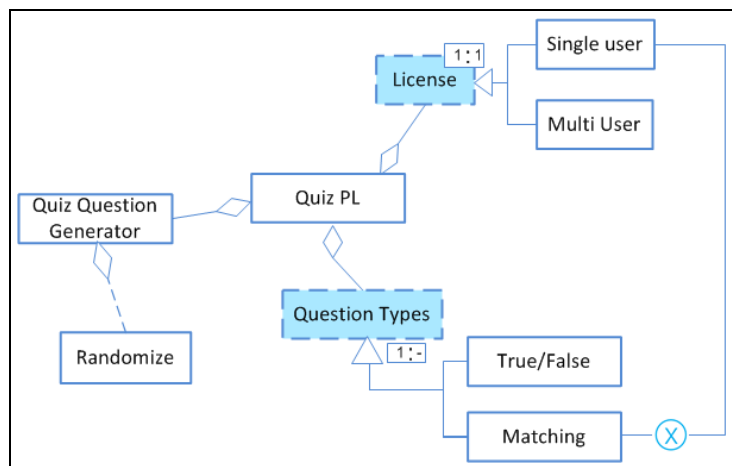


Figure 10.7: An Excerpt of the QPL System Perspective

```
Declaration(NamedIndividual(:Quiz_PL_System_Perspective))
ClassAssertion(:System :Quiz_PL_System_Perspective)
```

Next, we define the concrete feature *QuizPL* concrete feature, and assign it to the *Quiz_PL_System_Perspective* via the *belongs_To* object property:

```
Declaration(NamedIndividual(:QuizPL))
```

⁶⁰ The FAM Ontology can be downloaded from: http://wise.vub.ac.be/feature_assembly/FAM/FAM.owl

⁶¹ The complete QPL representation using FAM can be found at: http://wise.vub.ac.be/feature_assembly/FAM/FAM_QPL.owl


```

ClassAssertion(FAM:Concrete_Feature :QuizPL)
ObjectPropertyAssertion(:Belongs_to :QuizPL :Quiz_PL_System_Perspective)

```

Similarly, we also define the abstract features: *License*, and *Question Types*, which belong to the *Quiz_PL_System_Perspective*. Furthermore, the *Reports* feature is the same feature as the *Quiz Reporting* feature in the functional perspective (*Quiz_PL_Functional_Perspective*). This is shown by the axioms below:

- **License:**

```

Declaration(NamedIndividual(:License))
ClassAssertion(FAM:Abstract_Feature :License)
ObjectPropertyAssertion(:Belongs_to :License :Quiz_PL_System_Perspective)

```

- **Question Types:**

```

Declaration(NamedIndividual(:Question_Types))
ClassAssertion(:Abstract_Feature :Question_Types)

```

Similarly, we also define the concrete feature *Quiz_Question_Generator* that belongs to the *Quiz_PL_System_Perspective*:

```

ClassAssertion(:Concrete_Feature :Quiz_Question_Generator)
Declaration(NamedIndividual(:Quiz_Question_Generator))
ObjectPropertyAssertion(:Belongs_to :Quiz_Question_Generator :Quiz_PL_System_Perspective)

```

Next, we define the feature relations for each defined feature. Starting with *QuizPL*, it is mandatory composed of *License*, *Question_Types*, and *Quiz_Question_Generator*. This is defined via the object property *Mandatory_Composition*. It is also optionally composed of *Reports*; this is defined via the object property *Optional_Composition*, as shown by the following set of axioms:

```

ObjectPropertyAssertion(:Mandatory_Composition :QuizPL :Quiz_Question_Generator)
ObjectPropertyAssertion(:Mandatory_Composition :QuizPL :Question_Types)
ObjectPropertyAssertion(:Mandatory_Composition :QuizPL :License)

```

The next set of axioms define the options of the *License* feature, it also associates the feature with its corresponding cardinality, named *License_Cardinality*, a maximum and minimum cardinality is specified:

```

ClassAssertion(:Concrete_Feature :Multi_User)
ObjectPropertyAssertion(:Belongs_to :Multi_User :Quiz_PL_System_Perspective)
ClassAssertion(:Concrete_Feature :Single_User)
ObjectPropertyAssertion(:Belongs_to :Single_User :Quiz_PL_System_Perspective)
ObjectPropertyAssertion(:has_Option :License :Multi_User)
ObjectPropertyAssertion(:has_Option :License :Single_User)
ObjectPropertyAssertion(:has_Cardinality :License :License_Cardinality)

```

```

ClassAssertion(:FCardinality :License_Cardinality)
DataPropertyAssertion(:Max :License_Cardinality "1"^^xsd:string)
DataPropertyAssertion(:Min :License_Cardinality "1"^^xsd:string)

```

The next set of axioms defines the different types of questions supported for the *Question_Types* abstract feature. It is also associated with a cardinality individual named *License_Cardinality*, a maximum and minimum cardinality is specified:

```

ClassAssertion(:Concrete_Feature :Matching)
ObjectPropertyAssertion(:Belongs_to :Matching :Quiz_PL_System_Perspective)
ClassAssertion(:Concrete_Feature :True_False)
ObjectPropertyAssertion(:Belongs_to :True_False :Quiz_PL_System_Perspective)
ClassAssertion(:FCardinality :Question_Cardinality)
DataPropertyAssertion(:Max :Question_Cardinality "any"^^xsd:string)
DataPropertyAssertion(:Min :Question_Cardinality "1"^^xsd:int)

```

The next set of axioms defines the composition of the *Quiz_Question_Generator* concrete feature. It has an optional composition of the concrete feature *Randomize*, this is defined by the following set of axioms:

```

ClassAssertion(owl:Thing :Randomize)
ObjectPropertyAssertion(:Belongs_to :Randomize:Quiz_PL_System_Perspective)
ObjectPropertyAssertion(:Optional_Composition :Quiz_Question_Generator
:Randomize)

```

The next set of axioms defines the *excludes* dependency between the *Matching* feature and the *Single User* feature. The defined feature dependency is enriched with the use of the *Dependency_Description* annotation to document the rationale of the dependency. The *Dependency_Owner* annotation documents the stakeholder that defined the dependency.

```

ObjectPropertyAssertion(Annotation(:Dependency_Reason "A soft dependency based on
that Single user license is a free product with limited capabilities.
Sophisticated question types such as matching are not part of this free version.")
Annotation(:Dependency_Owner :Lamia) :excludes :Matching :Single_User)

```

As can be seen from the above example, representing a Feature Assembly model using the FAM Ontology is a straightforward process. Moreover, ontology editors simplify the above-mentioned process as they allow doing this process visually. For example, figure 10.8 shows defining this feature dependency and adding the *Dependency_Reason* and the *Dependency_Owner* annotations to the dependency assertion in Protégé.

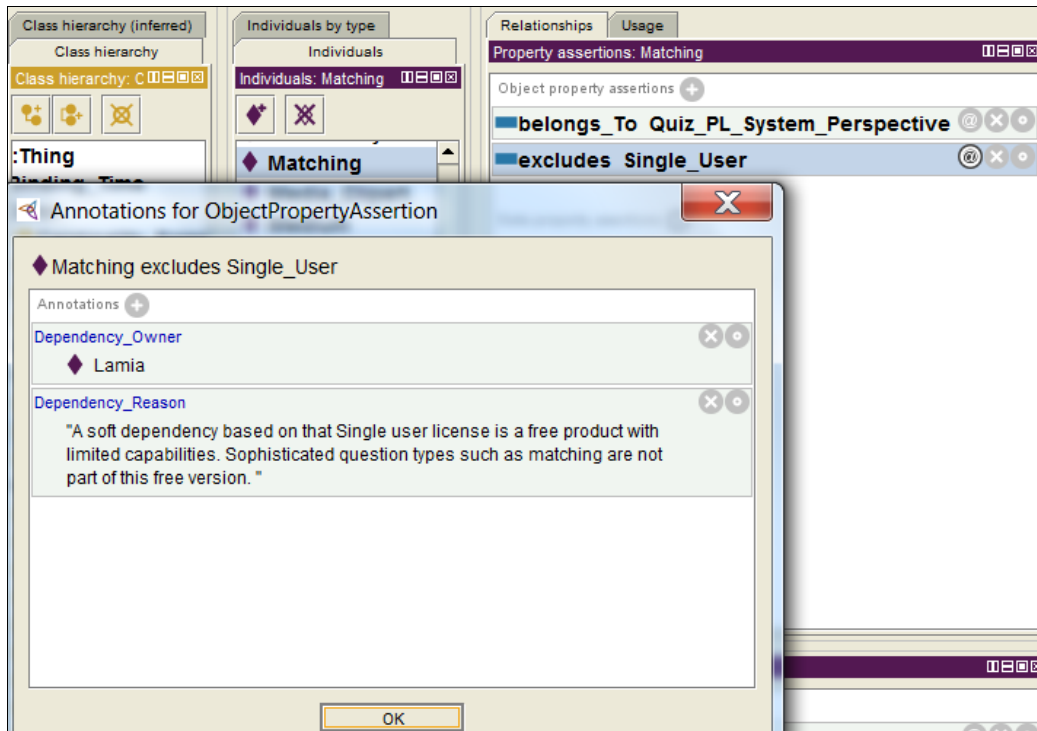


Figure 10.8: Example using of *Dependency_Reason* and the *Dependency_Owner* annotations in Protégé.

10.3 FAM Knowledge Manipulation

As already mentioned, the FAM Ontology is a *machine processable* knowledge model that defines the formal semantics for the Feature Assembly Modelling Language. Furthermore, with the help of DL reasoners new information can be inferred from the ontology and based on this some modelling errors can be automatically detected. Once populated with instances it becomes a single point of access for managing and manipulating the stored Feature Assembly models. As already mentioned, one of the merits of using the FAM Ontology is that it integrates feature assembly models created in different perspectives based on their intra-perspective dependencies. In general, two different approaches are used for finding information in large repositories: *browsing* and *querying*. We show below how each approach can be applied for the FAM Ontology based on general-purpose ontology tools. However, we also show how the two approaches can be combined in one FAM dedicated tool.

10.3.1 FAM Ontology Browsing

Browsing is used when somebody is looking for information but doesn't know exactly how to find it or whether it is available at all. In the case of FAM for example, developers may be looking for reusable features without knowing exactly which features they are looking for, or sales persons that need to know whether a certain feature exists for a certain system without exactly knowing the name of the feature. For this purpose, a general-purpose ontology browser could be used to allow navigating through the FAM Ontology. We illustrate this by using the JOWLBrowser (shown in figure 10.9). The ontology browser displays the ontology contents by means of classes, properties and individuals, which corresponds to FAM concepts, FAM

relations and FAM instances respectively. Additionally, the jOWLBrowser also allows for free-form search. A reasoner should be applied to the ontology (e.g., via Protégé) before actually browsing it in order to have inferred knowledge as part of the browsed ontology.

For example, figure 10.9 shows an example where a FAM ontology is explored for concrete features, and the details of the *Functional Quiz PL* concrete feature (which belongs to the Functional perspective) is requested. Using this browser users can search for features that satisfy a certain condition, for example, features that are variation points, features that are concrete; or features that have a specific property which can be for example belonging to a certain perspective, a feature to feature constraint (i.e. one of the feature dependencies excludes, includes, ..). Similarly, users can browse perspectives and limit their browsing to specific parts of the knowledge, for example perspectives that have a certain stakeholder, or a certain keywords and so on.

The screenshot displays the jOWLBrowser interface for the 'Feature Assembly Model Ontology for the Quiz Product Line'. At the top, there are tabs for 'Classes', 'Properties', 'Individuals', and 'SPARQL-DL'. Below the tabs is a list of ontology classes. The main area is divided into two panes: 'Treeview' on the left and 'Description of Concrete Feature' on the right. The 'Treeview' pane shows a hierarchy starting with 'Feature' and a sub-entry for 'Concrete Feature'. The 'Description of Concrete Feature' pane shows details for the 'Functional Quiz PL' instance, including its terms, disjoint with, and instances. A search bar is located at the bottom left of the interface.

Figure 10.9: Browsing the FAM Ontology for QPL using the jOWLBrowser – Showing Details for Functional Quiz PL Concrete Feature which Belongs to the Functional Perspective

10.3.2 FAM Ontology Querying

There are many scenarios where the users know what they are looking for. For example some users may want to search for all features that satisfy certain criteria. For example, for searching to find specific features, many criteria can be combined such as: the perspective,

involved stakeholders, and feature dependencies. The need for FAM Ontology querying stems from these needs. Querying for information allows users to find exactly what they are looking for, provided that they can formulate a search criterion. Users will use queries to find certain characteristics of the represented models in order to gain better insight or to solve existing problems (e.g., inconsistency or redundancy).

To illustrate querying for information in the FAM Ontology, the SPARQL query language was used. The SPARQL queries are executed against the populated FAM Ontology and returns a list of instance tuples that satisfy the query. Using the OWL2Query plugin in protégé, a SPARQL query can be constructed either visually or using SPARQL query language. In this section, we give some query examples to show some possible patterns of finding information in the FAM models using the FAM Ontology.

The users may be interested to know which features exclude each other. This can be answered by querying all the features that act either as source or as destination of an *excludes* property. To answer this quest, the following query can be used:

```
SELECT      ?Destination_Feature ?Source_Feature WHERE      { ?Source_Feature
<http://wise.vub.ac.be/Members/lamia/variability/Feature_Assembly/FAM.owl#excludes
> ?Destination_Feature . }
```

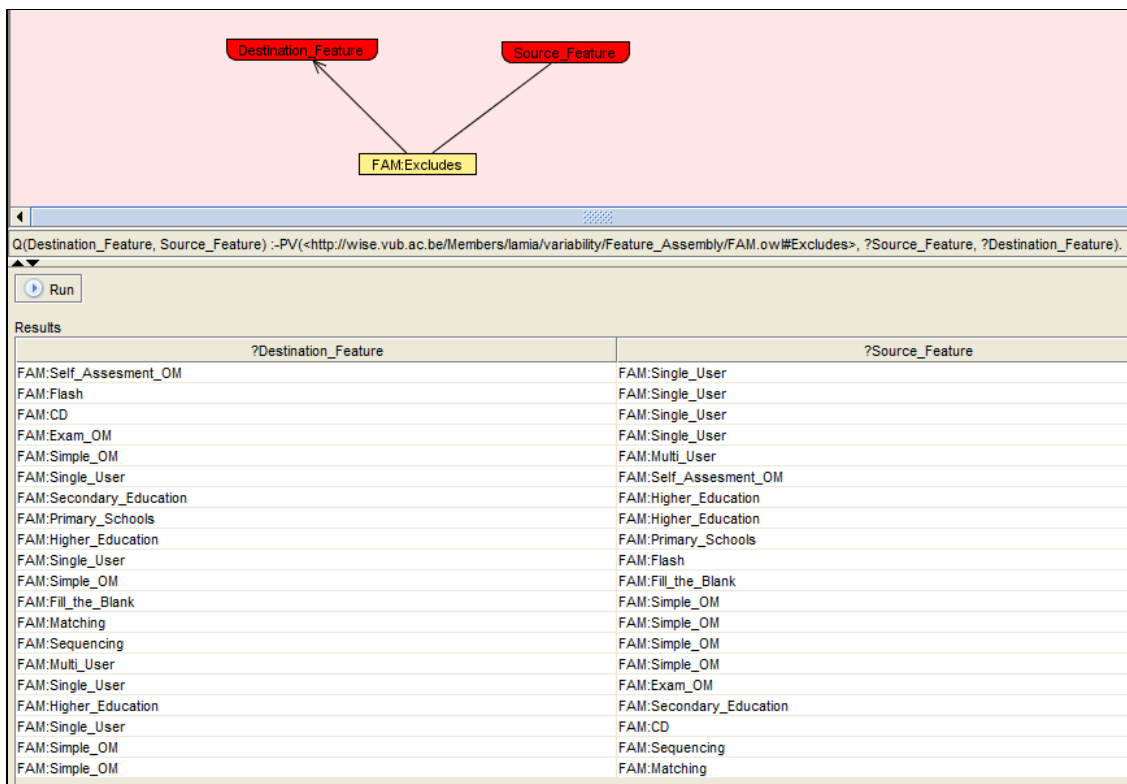


Figure 10.10: Querying for Features with an Exclude Dependency Using the OWL2Query Plugin in Protégé

Figure 10.10 shows the results of this query, and its visual representation using the OWL2Query Protégé query plugin.

We can further refine the above-mentioned query by asking for features that also belong to the system perspective:

```

SELECT  ?Destination_Feature  ?Source_Feature
WHERE{?Source_Feature
<http://wise.vub.ac.be/Members/lamia/variability/Feature_Assembly/FAM.owl#excludes
>
      ?Destination_Feature  .
      ?Destination_Feature
<http://wise.vub.ac.be/Members/lamia/variability/Feature_Assembly/FAM.owl#Belongs_
to>
http://wise.vub.ac.be/Members/lamia/variability/Feature_Assembly/FAM.owl#Quiz_PL_F
unctional_Perspective }

```

Additionally, more sophisticated queries can be created to ask for *optional information* within the triple patterns linking the data. For example, we may need to ask for the stakeholders involved with features that have an *excludes* dependency. At the same time, we do not want to miss the features which do not have a stakeholder assigned (via the `has_Stakholder` object property). To solve this, we define these parts of the query as *optional* (using the OPTIONAL SPARQL keyword) as shown below:

```

SELECT ?Destination_Feature  ?dst_stakeholder  ?Source_Feature  ?src_stakeholder
WHERE  {
  ?Source_Feature
<http://wise.vub.ac.be/Members/lamia/variability/Feature_Assembly/FAM.owl#excludes
>
      ?Destination_Feature  .
      ?Destination_Feature
<http://wise.vub.ac.be/Members/lamia/variability/Feature_Assembly/FAM.owl#Belongs_
to>
<http://wise.vub.ac.be/Members/lamia/variability/Feature_Assembly/FAM.owl#Quiz_PL_
Functional_Perspective>
      OPTIONAL  {
        ?Source_Feature
<http://wise.vub.ac.be/Members/lamia/variability/Feature_Assembly/FAM.owl#has_Stak
eholder>
          ?src_stakeholder}
      OPTIONAL  {
        ?Destination_Feature
<http://wise.vub.ac.be/Members/lamia/variability/Feature_Assembly/FAM.owl#has_Stak
eholder>
          ?dst_stakeholder }
}

```

Using the above-mentioned patterns, many different queries can be formulated to ask for different information concerning the software system modelled by the Feature Assembly Modelling language and represented by the FAM Ontology. Because it is difficult for users with no experience of SPARQL to formulate these queries, these queries can also be formulated using interactive search forms. This is described in the next section.

10.3.3 Dedicated Ontology Browsing and Querying

General-purpose ontology browsers have no awareness of the meta-model behind the information represented by the ontology. The ontology concepts or meta-model (i.e. classes, properties and instances) is what is actually being presented. In the case of Feature Assembly models it would be more useful if the information could be browsed in terms of *features* and *perspectives* rather than in terms of *classes* and *properties*. For this purpose we have implemented⁶² a dedicated ontology browser to allow users to visually and interactively

⁶² Credit for the implementation of the FAM Ontology browser and visualizer goes to our Bachelor student Jasper Tack. A running version of this implementation can be found at:

navigate through the ontology and find information. The *Feature Assembly Ontology browser* visualizes the represented information in terms of Feature Assembly models using the notations of the FAM language (see section 6.5 for the notations). Furthermore, for the sake of scalability the users navigate through the existing Feature Assembly models by clicking on the features of their interest to allow them to expand. This expansion on demand property of the visualization allows keeping the point of interest focused in large models (this was a recommendation distilled from our early prototype, see chapter 11 for more details) as opposed to expanding all the nodes at once up to a certain level.

Figure 10.11 shows the system perspective for the QPL where the *Reports* feature has been expanded. It is also possible to visualize the Feature Assembly models of more than one perspective by selecting the perspectives to visualize from the *Perspectives* tab as shown in figure 10.12. In this case the visualization allows to view the features common to more than one perspective (e.g. *Reports* in figure 10.12). Furthermore, the feature dependencies among the visualized features are also shown, therefore allowing to understand how the features of the different perspectives are linked to each other.

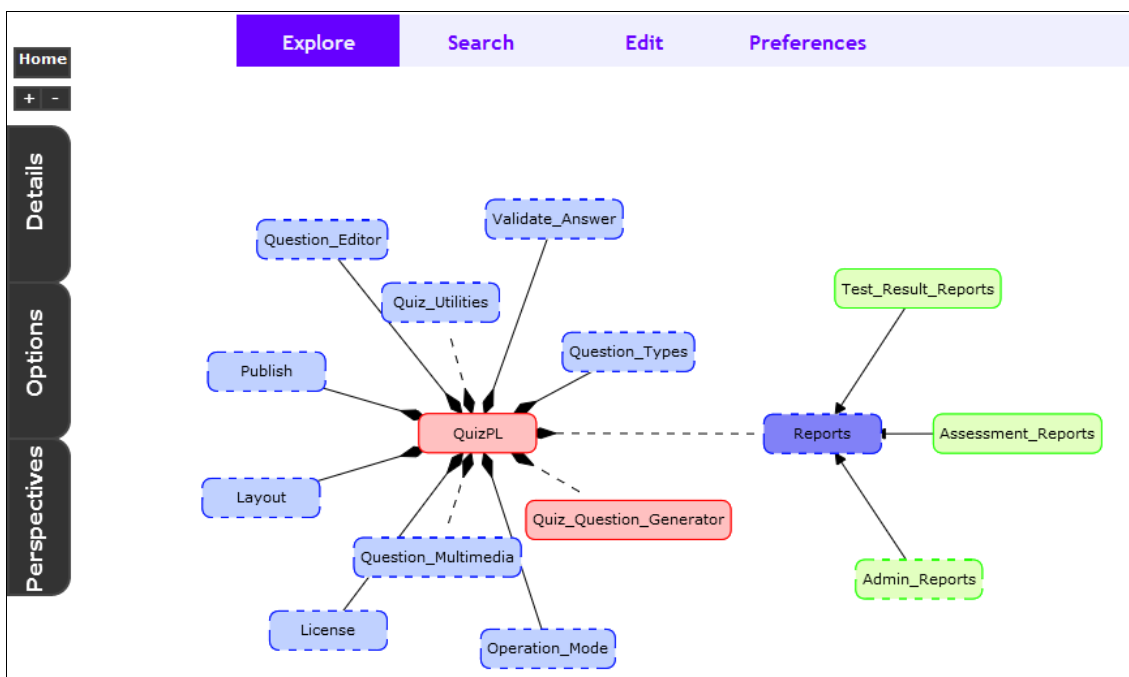


Figure 10.11: The FAM Ontology browser visualizing the QPL

http://wise.vub.ac.be:8080/FAM_FeaturePool/FPvisualizer.html. Note that in this version a larger set of dependencies are support than we currently do.

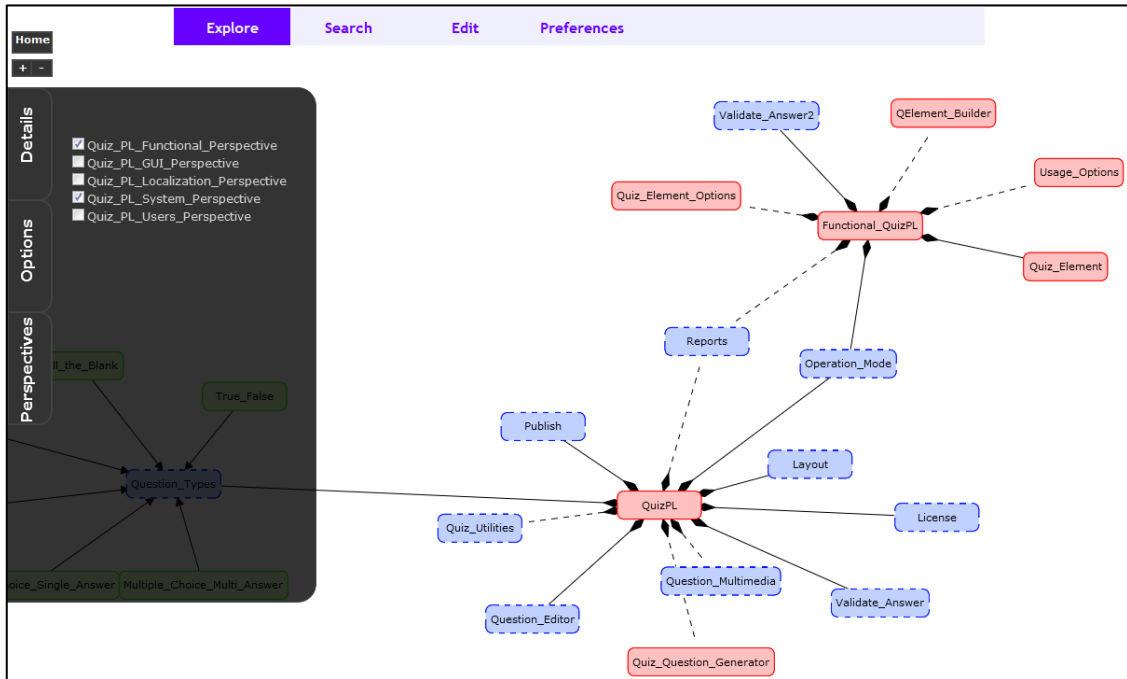


Figure 10.12: The *Perspectives* Tab allows visualizing the Feature Assembly models of more than one perspective at the same time.

The *Options* tab allows users to select the feature relations and feature dependencies that should be shown in the visualization, as shown in figure 10.13.

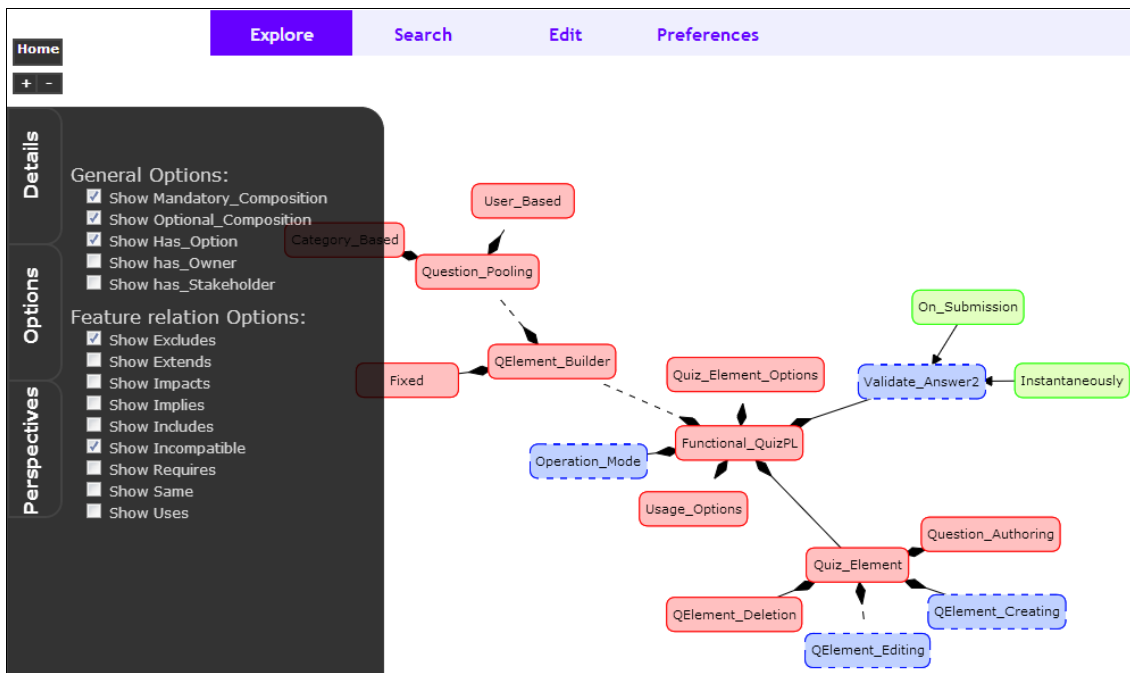


Figure 10.13: The *Options* Tab allows visualizing the Feature Assembly models selecting which feature relations and feature dependencies to view

For example, users may be interested in only viewing features with a composition relation, in this case only part of the graph that represent composition links (both mandatory

composition and optional composition) will be displayed. Likewise, users may wish to focus only on core features common to all products, in this case only features with mandatory composition relations could be displayed; or combine between viewing certain feature dependencies and certain relations. This allows users to concentrate only on the selected set of feature relations and features dependencies abstracting from all the details of the complete model. The *Details* tab displays the detailed information of the selected feature.

Additionally, the FAM Ontology browser allows users to search the ontology for features using certain criteria. The input of these criteria is enabled via a search form. Users can search for features containing some text, belonging to a certain perspective, owned by a specific stakeholder or used by specific stakeholders. Users can also search by the type of feature (e.g. abstract, concrete, or option), the type of relations and dependencies the feature has (e.g. optional, mandatory, ... or excludes, requires, ...) or the specific keywords associated with the feature. Any combination of the above criteria is possible. The search query is internally translated into a SPARQL query to derive the result from the FAM Ontology. The resulting features are associated with a *Details* button and a *Display* button as shown in figure 10.14. Figure 10.14 shows the search result of searching for a feature that has “Quiz” as part of its name, and is an abstract feature.

The screenshot displays the search interface of the FAM Ontology browser. At the top, there are four tabs: 'Explore', 'Search' (which is active), 'Edit', and 'Preferences'. Below the tabs is a search form with the following fields:

- Name:** A text input field containing the word 'quiz'.
- Perspective:** A dropdown menu with a list of perspectives: '--', 'Quiz_PL_Functional_Perspective', 'Quiz_PL_GUI_Perspective', and 'Quiz_PL_Localization_Perspective'.
- Type:** A dropdown menu set to 'Abstract_Feature'.
- Option Feature:** A checkbox that is currently unchecked.
- Standalone:** A checkbox that is currently unchecked.
- Has relation:** A dropdown menu set to '--'.
- Owner/Stakeholder:** Two dropdown menus, both set to '--'.
- Keyword:** A dropdown menu set to '--'.

Below the search form is a 'search' button. Underneath the search form, it indicates 'Page 1 of 1' and '2 item(s) were found'. The search results are displayed in a table-like format with two rows:

Details	Quiz_Layout	Options: 3	Perspectives: -
Display	Type: Abstract_Feature		Quiz_PL_Functional_Perspective
Details	Quiz_Uilities	Options: 3	Perspectives: -
Display	Type: Abstract_Feature		Quiz_PL_System_Perspective

Figure 10.14: The FAM Ontology browser's Search facility.

The *Display* button displays the visualization of the Feature Assembly model to which the resulting feature belongs, the resulting feature is indicated by highlighting it (here with yellow colour) as shown in figure 10.15 for the *Quiz_Layout*; the details of the feature are also displayed in a pop up window. This facility allows users to search for features specifying certain criteria and then navigating the models starting from these features.

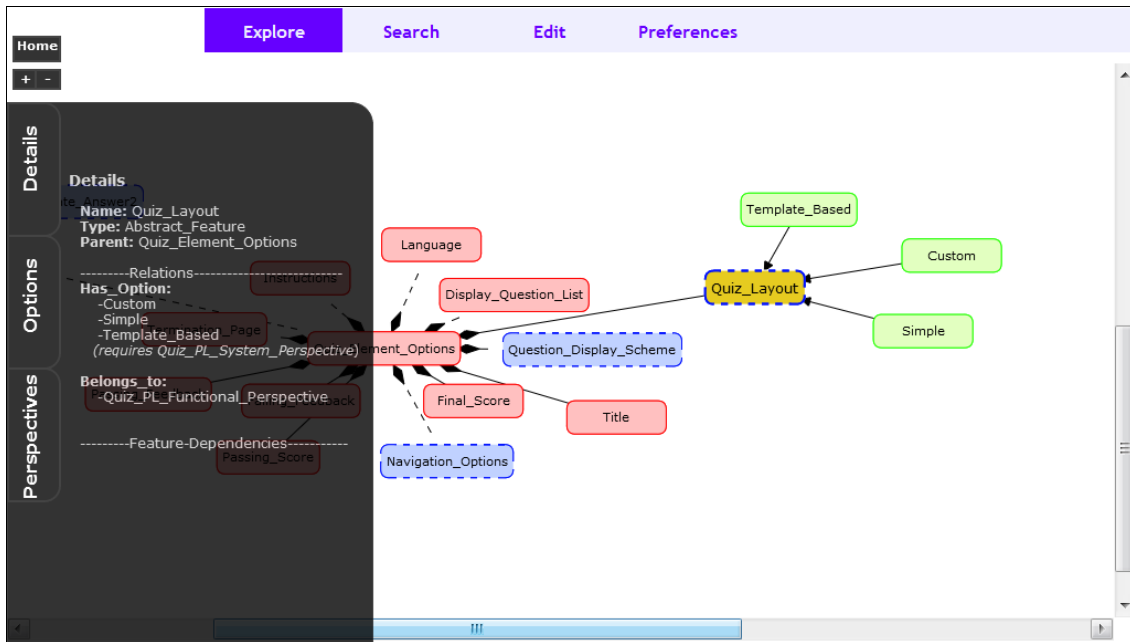


Figure 10.15: The FAM Ontology browser's Display facility.

These examples illustrate that the FAM Ontology browser allows users to interact with the information contained in the FAM Ontology without the need for ontological knowledge (and OWL knowledge); only the understanding of Feature Assembly notations is required. Also form-based search is simpler to use than writing full-fledged SPARQL queries. This shields users from the ontology behind the scene while gaining the benefits of using an ontology.

10.4 The Feature Pool Ontology Representation

As already stated in chapter 9, the Feature Pool is a repository of features with only the information essential for feature reuse. Revisiting the structure of the Feature Pool shown in figure 10.16, on the level of the ontology concepts (i.e. TBox) the FP Ontology shares the same concept structure as the FAM Ontology. Therefore, concepts related to capturing errors (i.e. Error class) and associated error detection SWRL rules are not part of the Feature Pool ontology. Thanks to the modular representation of the FAM ontology the Feature Pool Ontology could be easily extracted from the FAM ontology. Note that ontology reuse by ontology module extraction is a well-known technique in the ontology engineering domain for reusing existing ontologies to create new ones [Rector et al., 2005] [Doran et al., 2007]. Furthermore, on the instance level (i.e. ABox), one can consider the Feature Pool as a subset of the FAM Ontology, containing only the feature information that needs to be stored for enabling reuse (please refer to section 9.3.1 for an example).

The concepts that were extracted from the FAM Ontology include: *Feature*, *Perspective*, *Stakeholder*, and *Variability*. In addition to extracting these classes, the object properties that associate them were also extracted. The following properties were extracted: *Belongs_to*, *Composition*, *FTFC*, *Has_Option*, *Option_Of*, *Used_in*, *has_Owner*, and *has_Stakeholder*.

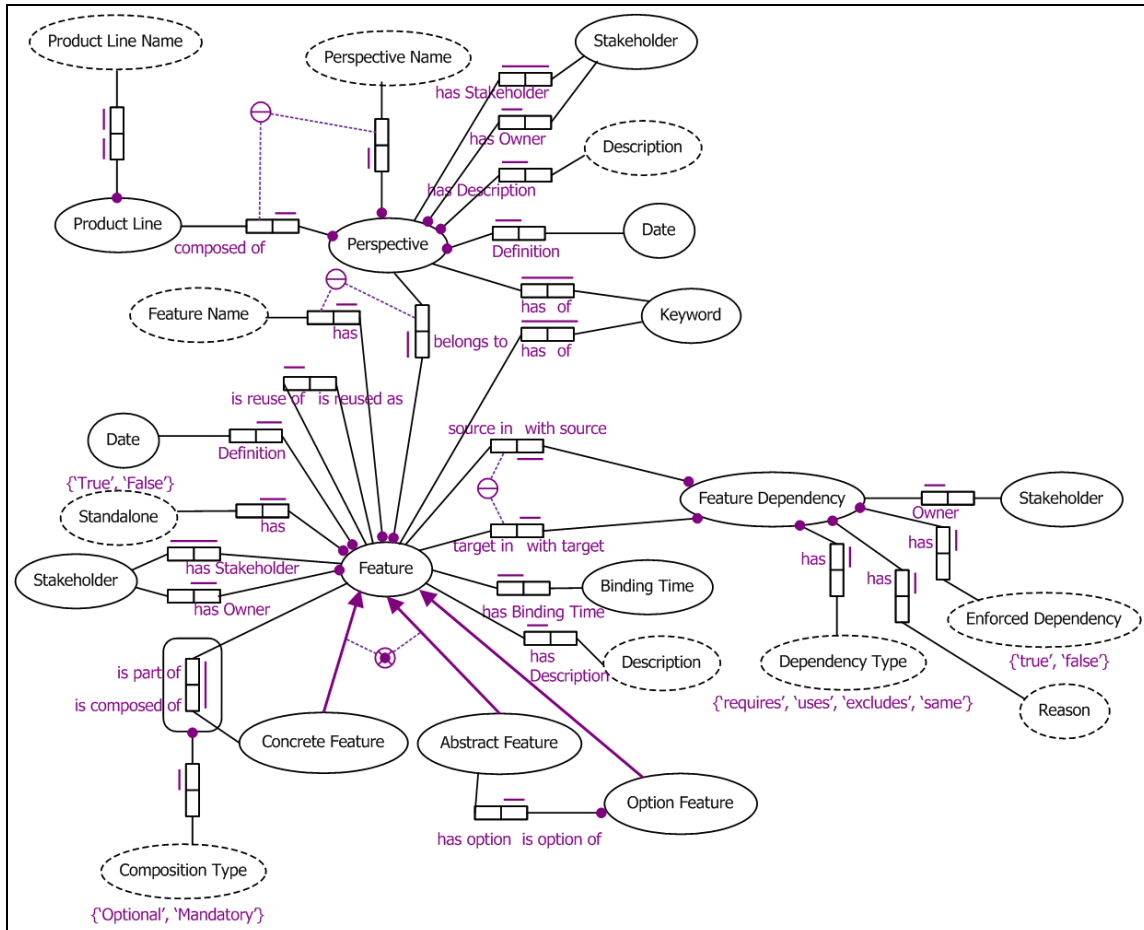


Figure 10.16: Feature Pool meta-model

Additionally, some new concepts (both classes and properties) needed to be introduced to the Feature Pool Ontology (FP Ontology⁶³). The FP Ontology contains two new classes `Product_Line` and `Keywords`. The `Product_Line` class refers to the product (lines) in which the features appear. This link is maintained via the perspective to which the feature belongs. The object property `has_Perspective` represents this link. It has domain `Product_Line` and range `Perspective`. `Keywords` are associated to features via the `has_Keyword` object property, which has domain `Feature` and range `Keyword`.

Figure 10.17 shows the FP Ontology in Protégé, showing the class hierarchy of the Feature Pool. Figure 10.17 also shows the usage of the `Product_Line` class, in this case there are two product lines defined in the feature pool (i.e. instances of the product line class) `Quiz_Product_Line` and `Exam_Product_Line`.

⁶³ The FP Ontology is given in appendix D, it can also be found at http://wise.vub.ac.be/feature_assembly/FAM/FeaturePool.owl

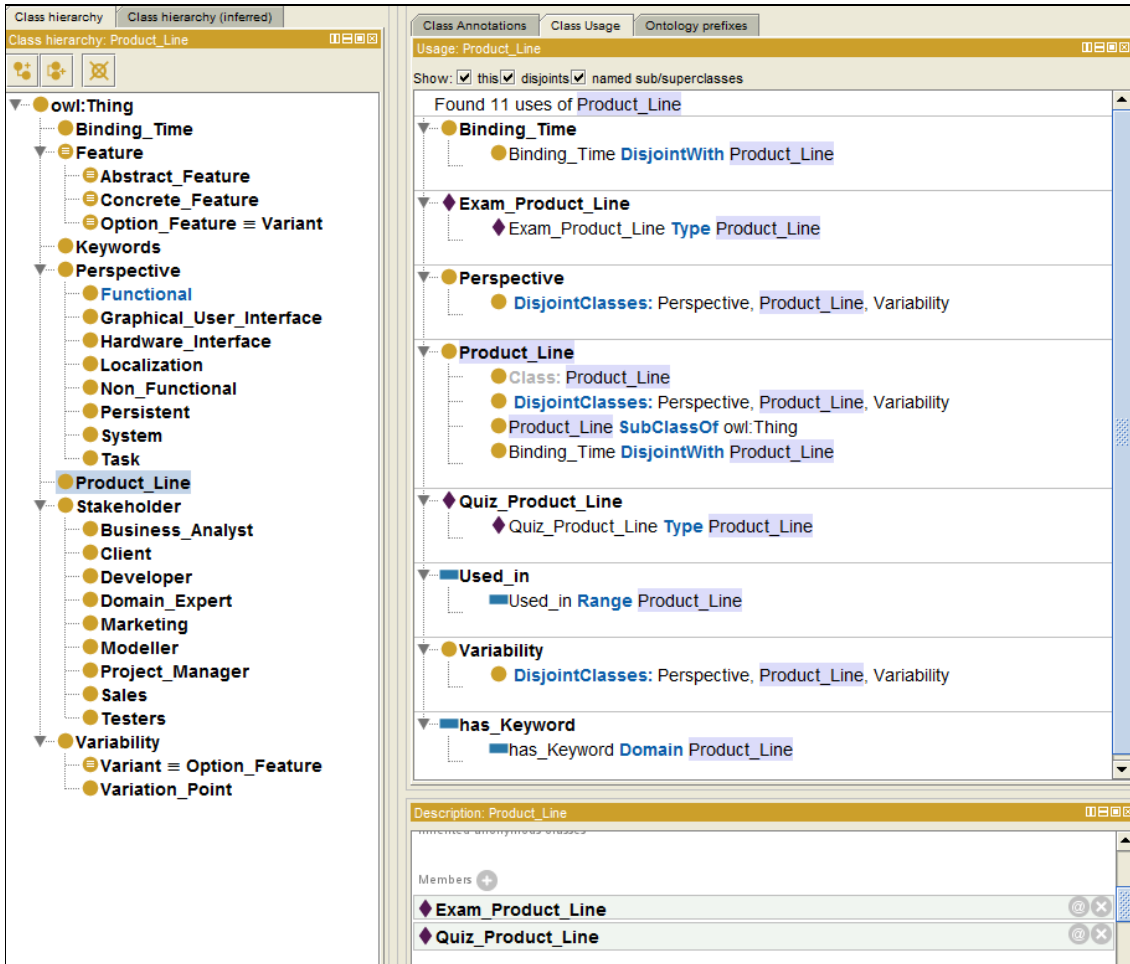


Figure 10.17: FP Ontology in Protégé, Showing Usage of the Product Line Class.

Furthermore, the fact that the FAM Ontology and the FP Ontology share the same concepts (TBox) allows us to use the dedicated Feature Assembly Ontology browser to navigate and search the contents of the Feature Pool.

10.5 Summary

In this chapter, we presented the Feature Assembly Knowledge Representation Framework which answers our third research question RQ2. Knowledge concerning the different features, their relations, and dependencies are represented by means of an ontology called the Feature Assembly Model Ontology (FAM Ontology). The FAM ontology provides a formal and machine processable representation of the Feature Assembly models introduced in chapter 6. The FAM Ontology was implemented using OWL DL, a decidable subset of the OWL ontology language. The FAM Ontology acts as a formal documentation store for information contained in the graphical Feature Assembly models. The ontology also allows using the power of DL for enforcing the feature assembly models formulation rules in order to guarantee the well-form ness of the created feature assembly models. Furthermore, the FAM Ontology defines a set of rules necessary to detect Feature Assembly modelling errors. We have formulated a set of rules that allow detecting four types of modelling errors: cycles in feature dependencies, inconsistencies between features dependencies, redundant feature

dependencies, and cardinality errors. We have also shown by means of an example how the FAM Ontology vocabulary can be used to represent individual Feature Assembly models.

We have also shown the power of such a representation for information retrieval. We have demonstrated two types of information retrieval methods, browsing for information and querying for information. We have shown by means of examples how complex queries can be issued to reveal information contained in the Feature Assembly models. Furthermore, we have shown how the technicalities of ontologies can be shield from the end-user by providing a dedicated browsing and querying tool for Feature Assembly Models rather than using general-purpose ontology tools.

Finally, we concluded the chapter by pointing out how the same knowledge representation technique was applied for the Feature Pool. The Feature Pool is implemented via the Feature Pool Ontology which is an OWL-DL representation of the structure of the Feature Pool. Furthermore, the Feature Pool may be populated with instances to provide a store for the feature information of more than one product line, thus allowing finding information for feature reuse. Actually, the Feature Pool Ontology was extracted from the FAM Ontology. Therefore, the Feature Pool can be browsed and searched using the Feature Assembly Ontology browser.

Chapter 11

Feature Assembly in Practice

So far in the thesis, we have presented the Feature Assembly approach for modelling, managing and reusing feature models. As already mentioned, the Feature Assembly approach tries to address some of the limitations of existing feature modelling methods. Furthermore, it allows for management of the knowledge of the feature models throughout the lifecycle of the product. Additionally, the Feature Assembly Reuse Framework aims at reusing existing feature specifications when modelling new systems or expanding current ones. In this chapter we present an industrial⁶⁴ experience of applying the Feature Assembly approach in an IT company. The main purpose was to conduct an exploratory study to validate the approach in an industrial setting.

11.1 Pilot Survey

Prior to applying our research in an industrial setting, we have surveyed the relevance of the feature assembly approach among 16 companies. We briefly presented the Feature Assembly approach by means of an elevator pitch during a gathering of software companies interested in software variability. Following that, we also had an individual 5 to 10 minutes discussion with interested representatives. The discussions targeted the way they handle variability in their systems, how they explicitly model it (if any), how they manage the information in their models, what type of information they need to know about their features, and the importance of the concept of modelling with reuse for them. After the gathering (which lasted about 2 hours), the participants were asked to fill in a questionnaire, which (among other things) rated the approach based on its relevance for the company. A 5 point scale was used: bad, weak, average, good, and excellent. Note that during this gathering also other variability solutions were proposed. Out of the 16 companies three companies gave a rating of excellent, eight gave a rating of good, one gave a rating of average and four companies did not fill the questionnaire, leading to an overall rating of good. This shows that there is a good interest of companies in the presented technique.

This indicates that companies developing multiple related software products or products having different variants are faced with many challenges. The discussions also revealed that there is a need in companies for support to manage the *dependencies* between their different features, modules or components, not only during analysis and design time but during the complete lifetime of the product. Most of the companies at the gathering were also interested in maximizing their reuse possibilities. Efficient reuse of already existing assets is a

⁶⁴ The findings presented in this chapter are based on the author's experience in the VariBru industrial project, in which periodic meetings are held with companies to communicate the consortiums research to industry.

major issue for them. We also note that these two issues are related, i.e. reuse is affected by the feature dependencies and vice versa.

It is agreed upon both in academia and in industry, that analysis and design are often underestimated when developing new products [Van Ommering and Bosch, 2002] [Codenie et al., 2009]. The impact of good design is obvious, yet good practice remains a challenge. Furthermore, it was found that in small and medium scale companies variability is not planned beforehand but actually evolves with time due to the expansion of the software to serve more customers or due to the need to customize some features to meet the different needs of different customers. In these situations, a poor product design may create problematic situations as the software becomes difficult to extend, becomes extremely complex and unstable, and most of the company's time is spent on bug fixing, maintenance and testing. Therefore for these companies a (processable) knowledge model that allows to efficiently keep track of the commonality and variability of features in their products looks promising. Furthermore, most of these companies work in a specific business sector, i.e. a single domain but with varying customer requirements. These companies can benefit the most from the reuse opportunities offered by the Feature Assembly Reuse Framework.

In the rest of this chapter we present our experience on using Feature Assembly in an industrial setting based on the work we did with a medium scale software company, ANTIDOT, working in the domain of web-based IT solutions and services for corporations, companies and associations.

11.2 ANTIDOT Experience Report

ANTIDOT was one of the companies that participated in the above-mentioned survey. The company was facing the problem of “products expansion”, and could immediately see some potential in the Feature Assembly Reuse Framework to help them keep track of all the existing as well as new features in their products. Particularly they wanted to keep track of all the different features they deliver to their customers in order to make the best benefit of reusing their existing features. These features were in their code base; they adopted an opportunistic reuse of features/code which made it difficult to keep track of the dependencies and existing variances of a certain feature at some point in time. Additionally, it was important for them to be able to analyse the different relations between the different features within their product, and in particular the feature dependencies because these affect greatly how they can customize the final product.

Furthermore, the company was reengineering their core product, a content management system (CMS), which they customize for many customers. They realized that although they had not planned variability beforehand, it found its way into their product over time. Moreover, after over 10 years working in the domain they gained a lot of experience in that domain. Based on this experience, they now could identify where to incorporate variability in their product, in order to make it more configurable, therefore reducing the development time/cost and meet their customer needs. Additionally, they wanted to investigate how variability modelling could help them in explicitly planning and representing the variability in their CMS product. Additionally, they were interested in the possibility of systematic reuse of the CMS features in their other products. Variability and commonality of features of the CMS and their dependencies was becoming a headache for them as the number of possible products grew due to continuously adding new features or feature variants.

As researchers, our goal was to validate our research results in a real world situation; therefore, we have formulated a set of questions that are oriented to measure the relevance of our Feature Assembly approach for the company. These questions can be applied to companies

that have some form of variability in their products, and which did not yet apply a feature modelling technique.

- Q1. Is the Feature Assembly modelling approach expressive⁶⁵ enough to deal with their variability modelling issues, i.e. does the company see added value in adopting this variability modelling approach?
- Q2. Does the company have a problem of concealed information (i.e. information hidden in code, paper documents, or in the heads of the developers)? Does the Feature Assembly Knowledge Management Framework help resolving this problem?
- Q3. Can we promote reuse of features (specifications) early in the development cycle? Does the company believe that this will make a difference for the development cost?

The study aimed at finding answers to these questions. Answering these questions should help us gain better understanding of the approach's feasibility as well as its limitations.

11.2.1 Method Adopted

In order to find answers to the questions formulated in the previous section, we were of the opinion that it was better that members of Antidot did the actual modelling of their product rather than us doing this. We had five meetings with two members of the company. In the first two meetings we introduced the Feature Assembly Modelling approach, Feature Assembly Reuse Framework (i.e. the concept of a Feature Pool for feature reuse) and the Feature Assembly knowledge manipulation. As an example, we created some partial feature models for one of their products. In the following three meetings we discussed the models they created for their system. Additionally, we explicitly asked for comments on the modelling approach, the Feature Assembly Knowledge Management prototype we created for testing the approach, and the feasibility of the Feature Pool approach.

The two participating members of the company held the roles of CEO and also Senior Project Manager, and senior developer (he also plays the role of the designer). The first two meetings lasted for one hour each; the next three meetings lasted for two hours each. In the next section we provide more details for the different aspects of the FAM approach that we considered.

11.2.2 Feature Assembly Modelling Technique

As already mentioned, the first meeting was dedicated to introducing the concept of variability modelling. The concept of “modelling variability” was also quite new for them, although they realized variability in their products. We introduced the Feature Assembly Modelling technique and the different modelling notations used. We also gave an example of how the modelling approach can be used in practice. The presented modelling technique and the examples we gave them were based on our publications [Abo Zaid et al., 2010, a] [Abo Zaid et al., 2010, b] which contained an extensive set of feature dependencies, the original set of feature dependencies were: *incompatible*, *excludes*, *uses*, *requires*, *extends*, *includes*, *impacts*

⁶⁵ According to [Wand and Weber, 1993], expressiveness means *completeness* and *clarity*. Completeness means that a modelling language has all the constructs that allow modellers to represent the domain information. Clarity means that the language is free from *construct redundancy*, *overload* and *excess*.

and *same* (we later reduced this set to the dependencies listed in section 6.5.3 based on the findings of this study and different discussion with other researchers about their usefulness).

We were also interested in understanding their domain and how they keep track of information in their current settings. They mentioned that they keep track of their modules and their functionality by means of an *excel file* that lists their modules and their functionality (an excerpt of this file is shown in figure 11.1).

We used this file as a starting point for our next meeting where we discussed the use of the Feature Assembly modelling techniques to represent their CMS. We also made some feature assembly models that represent parts of their product in order to initiate the discussion and which they could use as starting point for further modelling.

	A	B	C	D
1	Dashboard	Dashboard	Listing of latest activities within the admin sorted and grouped by date	
2				
3		Dashboard	Filter on user	
4	Pages	Structure Editor	Reorder nodes and cancel	Node Manipulation
5			Show nodes properties	Node Manipulation
6			Nodes and save	Node Manipulation
7				Node Manipulation
21			Recent activities	Node Manipulation
22			Search	
23		Page Properties	Save data	Page Property
24			Publish	Page Property
25			Delete	Page Property
26		Page Editor	Reorder	
27			Delete element	
28			Create draft	Draft
29			Add Headings	Add Resources
30			Add textblock	Add Resources
31				

Figure 11.1: Excerpt of the information in the Excel File containing the CMS specifications

During our meeting we used a pen-and-paper approach for creating the models (or rather modifying the created models). With these example models, they quickly got engaged into the modelling process, and started to make adjustments to the models we created. For us this was very positive, as it showed that the modelling notation and semantics we used was intuitive and easy to understand. Figure 11.2 shows an example of the annotations and corrections they made. Furthermore, they quickly understood the concept of perspective as illustrated by the fact that they remarked that some of the features we modelled in the System perspective were actually belonging to the Task perspective.

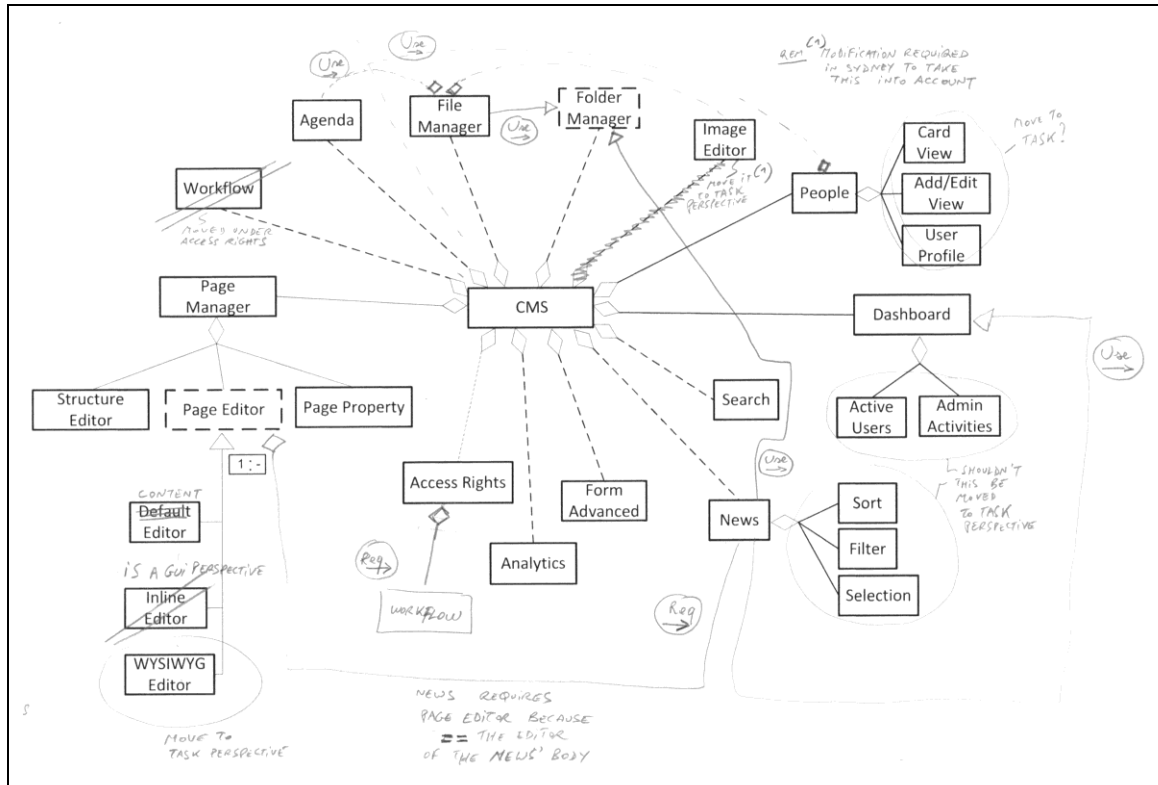


Figure 11.2: Excerpt of Comments on CMS's First Models

Together, we identified the following potential perspectives: System perspective, Task perspective and GUI perspective. Next we asked them to do the modelling⁶⁶ of their system, in order to investigate how easy/difficult the modelling process would be. In the following meetings, we discussed their models, answered their modelling questions, and collected their comments on the ease of use and intuition of the modelling approach. They reported that to analyse and model one major module of the CMS, it took one person about two hours and a half. This resulted in a model with 28 features and 21 connections between features (14 feature relations and 7 feature dependencies). In total, three persons were involved in the modelling of the CMS. An issue was the learning time for the notations used, although they appreciated the similarity with the UML notations (as they are using UML for system modelling). We report their remarks on their experience (after they had done their modelling homework):

- R1. Some features needed many feature dependencies and this was cumbersome to specify.
- R2. At first, the distinction between some feature dependencies was not always obvious and this initiated a discussion with other members to decide which one to use (e.g., 'uses' versus 'requires')
- R3. They were wondering at which level of detail they had to model.
- R4. It was not clear how they could specify external features/components.
- R5. Sometimes they found it difficult to decide which perspective to use for modelling certain features.
- R6. It was not clear if and how they could model "different versions of the same feature".

⁶⁶ We provided a Visio stencil that contains the notations used in the Feature Assembly Modelling Language.

Some of these remarks are due to the lack of experience with the Feature Assembly modelling technique and the lack of good documentation for the method (e.g., an elaborated user guide), such as remarks R2, R3, and R5. Also remark R5 was because they assumed that a feature should only belong to one perspective, which is not the case, a feature may have different faces, and therefore the same concept represented by the feature may belong to more than one perspective; the *same* feature dependency should be used to explicitly mark features defined in different perspectives that refer to the same feature. For them remark R1 triggered some important modelling questions: “Is it required to model all dependencies?” and in general “How can dependencies be minimized during design in order to eliminate the coupling between the features as much as possible, because high coupling will reduce the reusability?”. It actually turned out that the different features of their product were more coupled than they had expected. Remark R4 and R6 revealed some shortcomings of the Feature Assembly modelling approach, as the method currently doesn’t provide support for this. Remark R6 was actually interesting because it lead us to point them to the *extends* feature dependency to relate a feature to another version of it (which is superior to it). Currently, the Feature Assembly Modelling technique treats all features (external and non-external) similarly. Also it does not support “versioning” of features, in FAM different versions are different features. These issues should be considered in future work.

We have also noticed that after the second modelling meeting, the team was already comfortable using the modelling technique, capable of making decisions concerning the feature types and their dependencies. They were also comfortable using the term “feature” to refer to their system capabilities. We then asked them for the *expressiveness* and *ease of use* of the modelling technique. Specifically we asked:

MQ1. Did the concepts allow you to explicitly model all the information you wanted to express?

MQ2. Did you find the modelling concepts easy to understand?

MQ3. Did you experience some redundancy in the provided modelling concepts?

MQ4. Was it difficult to choose which modelling concept to use during the modelling practice?

MQ5. Did you find the modelling notations used appropriate for expressing the modelling concepts they represent?

MQ1 asks for the *completeness* of the modelling concepts provided, the team confirmed that the concepts were adequate to represent the variability and commonality in their CMS.

MQ2 checks for the ease of use and appropriateness of the presented modelling concepts for modelling features and their variability and commonality. Both team members confirmed the ease of use of the concepts to model features (abstract features and concrete features) and their relations. Yet they had some doubts about the feature dependencies, it was not always clear which dependency to use.

MQ3 checks for *construct redundancy*. The team confirmed that the features and relations concepts were adequate to represent the variability and commonality, while they could experience some redundancy between the feature dependencies (for example, *incompatible* and *excludes* were similar for them).

Answering MQ4, they indicated that the difference between the concepts of features and feature relations was obvious while again they had some problems with the feature dependencies. They identified their potential usable set of feature dependencies as (requires,

uses, excludes, extends and same); it was only this set that they used in their models. The *extends* dependency they only found relevant to indicate two versions of a feature. We agreed to substitute that with meta-data associated to the feature, therefore a *definition date* was added as part of the feature meta-data.

Answering MQ5, the team confirmed that the modelling notations used were intuitive and easy to understand; the similarity between UML notations and Feature assembly notations to model feature relations was appreciated.

Furthermore, the experience has confirmed the following merits of adopting the Feature Assembly modelling technique:

1. Feature Assembly let them reconsider their “features” in order to increase the modularity of the software. Using the Feature Assembly Modelling technique, dependencies between features became more visible and they could use this to improve the design for achieving a lower degree of coupling between modules/components at the code level.
2. Explicitly modelling variability and commonality triggered new potential variation points. As a consequence, more variability could be planned in the next version of the product.
3. Documenting and understanding the feature dependencies helps them in better defining their test scenarios, as the feature dependencies are reflected as module dependencies in the code.
4. Feature Assembly models help them better identifying the impact of changes in features.
5. The system perspective provides a better view on the important features of their product, providing a different level of abstraction and understanding of their system.

The team also reported that Feature Assembly Models helped them with *understanding and managing the evolving variability of their product over time*. Moreover, the team already uses UML models to model their system, but with Feature Assembly a different level of abstraction and understanding of their system was achieved. Modularity of their features was made more obvious and let them consider improving their design for achieving better modularity.

11.2.3 Feature Assembly Knowledge Manipulation

In order to justify the usefulness of readily finding information offered by the Feature Assembly Knowledge Management Framework proposed in this thesis, we asked the team to test our Feature Assembly models knowledge manipulation prototype in order to analyse and find information concerning the models they defined. We wanted to verify if there is a need for using the reasoning and rule support provided by an ontology or is storing, retrieving and visualizing the models enough. For this a first prototype⁶⁷ was used. This prototype was not

⁶⁷ Credit for this implementation goes to Tom Puttemans who implemented the Feature Assembly Explorer prototype, which provides an interactive visualization for the Feature Pool represented using the Feature Assembly Modelling technique. For more information on this implementation, please refer to [Puttemans, 2011]. A running version of this implementation can be found at: <http://wise.vub.ac.be:8080/FeaturePool/>

based on an ontology (as described in chapter 10) but on a relational database⁶⁸. However, the database structure was a one to one mapping of the FAM ontology defined in chapter 10. The prototype is web-based, visualizes feature assembly models to allow users to navigate visually through the models in order to find information. Furthermore, the prototype allows users to search (i.e. query) for information based on the feature names, feature description, feature type, and perspective name. Additionally the features belonging to a specific perspective can be listed. Figure 11.3 shows these different search possibilities. Forms were used, opposed to letting users formulate their search queries because forms are easier to use. Also a tag cloud was provided to enhance the searching via tags. The tag cloud clearly indicates the popularity of the tags used to mark features.

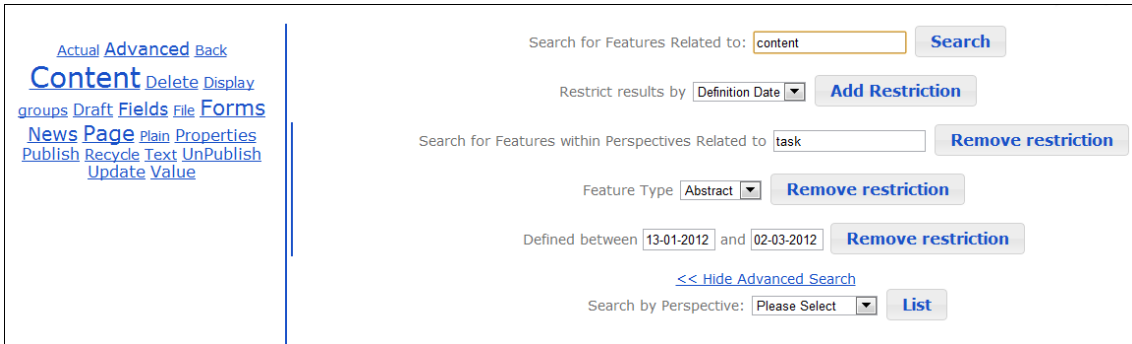


Figure 11.3: Screenshot showing how Information can be found in Feature Assembly Models - Applied to the models of Antidot.

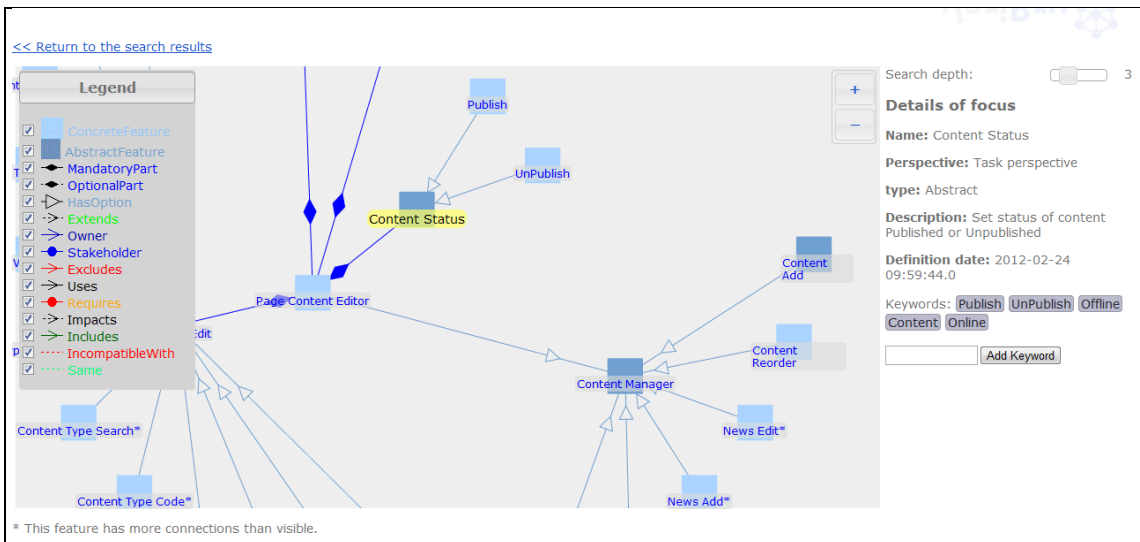


Figure 11.4: Screenshot showing how Feature Assembly Models are visualized allowing users to interact with the information contained in the models - Applied to the models of Antidot.

Figure 11.4 illustrates how users can visually interact with their Feature Assembly models. As shown in figure 11.4, the level of details can be changed via controlling the depth (of the decomposition showed). Furthermore, as different stakeholders are interested in

⁶⁸ At the time this study was conducted the FAM ontology and the Feature Assembly Ontology browser were not yet implemented. Feedback from this study has helped us define the capabilities of the Feature Assembly Ontology browser presented in section 10.3.3.

different parts of the information, we allow users to select/deselect the type of information they would like to view (in the legend at the left side).

The team of Antidot confirmed that providing a visual navigation mechanism for inspecting the models was indeed useful. Furthermore, allowing users to visually interact with the Feature Assembly models is useful when tracing a certain feature for its relations or dependencies. In their case, they had some features that represented the backbone of their system and which they found very useful to inspect using the prototype. This functionality is particularly important when more than one person is involved in the modelling (in their case three persons were involved). Also, they reported that being able to control the depth of display for a model during visualization is indeed useful for providing different levels of detail, although they preferred an expand-on-demand⁶⁹ rather an expand-all scheme when navigating through the feature assembly models. The team also recommended adding some important meta-data to the information stored. For example, they recommended adding a *description* for each perspective and a *definition date* for the features. A definition date could also help them overcome the lack of versioning support for the features mentioned in the previous section (we actually updated our prototype to include this and let them test it again).

11.2.4 The Feature Assembly Reuse Framework

As already mentioned the company was in the phase of reengineering their product. Among the discussions we had was the discussion of the applicability of the Feature Assembly Reuse Framework for reusing already specified features in the design of new products. Being a small company their reuse schema was based on the reuse of components and code at an implementation level. Reuse at a design level was not given too much attention yet. Introducing them to the concept of “reuse at a design level” has actually led them to reconsider the modularity of their features to enable more reuse opportunities. Furthermore, they agreed that considering reuse at a design level is important to promote component reuse rather than code reuse. Furthermore, two aspects were distinguished, “design for reuse” and “design with reuse”. To promote “design for reuse” the following guidelines were identified:

1. Identify which features are candidate standalone (i.e. consolidated and independent) features.
2. Analyse which of the feature dependencies are essential and should be enforced for these features.
3. Improve the models such that the feature dependencies between standalone features are minimized.
4. Use the meta-data to describe these features, in order to be able to easy retrieve them later on, in particular by the use of tags. Restricting the tags to a specific set (e.g. using a predefined set of keywords) was not recommended, but rather a growing pool of tags was advised.

To promote “design with reuse” the following requirements were identified:

1. A good search mechanism is needed to identify already existing and reusable features.
2. The need to invest time in carefully modelling (existing) software features.

⁶⁹ This remark has lead us to define an expand on demand scheme for the feature models visualized in our Feature Assembly Ontology browser

11.2.5 Discussion

We can conclude that the work done during this evaluation, as well as the discussions held, confirmed the value of the presented approach; it also revealed interesting future work (discussed in chapter 12).

The presented study clearly answered our research questions stated earlier, the company clearly stated that they see added value in applying feature analysis and modelling to their product(s). Furthermore, the time needed by Antidot to learn to use the Feature Assembly Modelling technique was quite short. The company was also very positive on the ease of use and intuition of the modelling concepts and notations. They reported no problems with the understandability of the modelling concepts (except for some of the dependencies). They have also evaluated the expressiveness of the presented Feature Assembly modelling technique through explicitly answering some questions concerning the modelling concepts and the notations used (MQ1-MQ5). This answers our first question (Q1), and also gave us some insight on improving our technique (by reducing the number of feature dependencies).

Feature Assembly knowledge manipulation (which is part of the Feature Assembly Knowledge Representation Framework) was also appreciated for providing an interactive medium for finding information about features in the Feature Assembly Models. For this to payoff, the company has to enforce a strict policy for adding meta information (e.g., feature description, feature keywords, stakeholders involved, customers who have this feature, etc.) and therefore making it available for later. From the discussions we had it was also clear that not all stakeholders need the same detailed level of information. For example, developers are interested in all levels of details for the modules they are responsible for, but for other modules they are only interested in the feature dependencies. It was clear that even this small company does have a need to unlock information implicitly available inside the company (Q2).

The presented study only provided a partial answer to our third research question considering feature reuse (Q3). Feature Assembly modelling allows making more modular designs. Furthermore, the Feature Assembly Framework helps efficiently retrieve features for reuse. Therefore we may say that it increases the chances of successful reuse inside the company, therefore increasing the chances of reducing development cost. However, actual reuse can only be achieved while developing a new product. This has not been performed during the study. Therefore, it was not possible to answer R3 with complete certainty.

11.3 Threats to Validity

As we only validated the approach with one company, it may be possible that experiences in other companies could be different. However, the company was unknown to the researchers before the study was started and the company also didn't have any reason to favour the approach or the researchers. Therefore, we can state that the results obtained are rather objective.

The fact that the company is a small-scale company may have had an impact on the results. As already mentioned, the company has not been using the concepts of variability modelling before, neither the concept of "feature" to describe their product capabilities. This may have affected the results in two different ways. First, introducing a new modelling technique may have introduced some learning time (which was indeed the case). Secondly, because Antidot has not used a variability modelling technique before they cannot compare the ease of use and expressiveness of Features Assembly to other feature modelling techniques.

The study was done in a rather informal way, i.e. using meetings and discussion. We believe that this is justified for a first (pilot) validation study, as the major purpose was to obtain as much spontaneous feedback as possible. In later case studies and experiments, a more rigorous approach will be used.

11.4 Summary

In this chapter, we first presented our experience in presenting the Feature Assembly approach to software companies. We also presented our experience in actually applying the approach in a medium scale software company. This exercise was fruitful in many ways. Firstly, it gave us some insight on how companies work and what their challenges are concerning specifying and managing the continuous growth and variation of their products. Secondly, it has clearly shown the importance of (the often underestimated) modelling of software, in particular variability.

This case shows that modelling software using the Feature Assembly Modelling technique improves the understanding of the features that compose the software and their relations, and provides a detailed overview of their contribution to the variability of the software. Additionally, it provides a good overview on the modularity of the software and the degree of coupling (represented by the feature dependencies). Moreover, the principle of perspectives for the separation of concerns has helped focussing on one specific point of view at a time. In particular, the system perspective was considered important because it gives an overview of the main software capabilities. This experiment also revealed the need for planning reuse and improving reuse opportunities by considering reuse early in the design process.

Being able to interact with the information contained in the feature assembly models was also a very important issue for the company. Models are important to gain a better understanding of the software, but being able to retrieve information from these models at different phases of the software is also important.

Chapter 12

Conclusions and Future Work

In this chapter we conclude the work presented in this thesis, we first start by summarizing the work presented. Next we highlight the main contributions of the thesis. We conclude with exploring possibilities for future work.

12.1 Summary

Introducing variability into software supports the development of many different but related software products instead of just one. However, it also raises the complexity; therefore introducing variability should be done carefully (i.e. modelled) in order to keep the complexity of the variable software under control. In this thesis we have presented the Feature Assembly approach for feature modelling and information management of software variability and commonality. This thesis tackles the following research questions (see figure 12.1, please refer to section 1.4 for more details).

RQ1: How can variability and commonality modelling in today's large and complex systems be supported by addressing current challenges and limitations?

RQ2: How can the knowledge in feature models and features be captured and unlocked?

For answering the above mentioned research questions, we have adopted a design science approach. The result being an approach that allows modelling software variability and commonality, and that allows for efficiently representing this information and allowing users to inspect and query the models. The solution we proposed, the Feature Assembly approach, is an integrated approach providing solutions to all problems addressed. We now summarize the steps taken in the research and the artefacts developed.

12.1.1 Steps in the Research and Artefacts developed:

In order to do achieve our solution, the Feature Assembly Approach, we first identified the challenges in modelling software variability taking into account the currently available modelling approaches seeking answers for the first parts of our first and second research questions **RQ1.1**, **RQ1.2** (see figure 12.1 for details). Our study has resulted in the identification of a number of limitations (see figure 12.1) of current feature modelling techniques: difficulties in identifying features and using the modelling technique in practice; semantic ambiguity of the modelling concepts used which resulted in poor expressiveness of the modelling notations; lack of scalability support and limited reuse opportunities of

previously modelled features. These limitations are shown in figure 12.1 indicating how they relate to the research questions raised in this thesis.

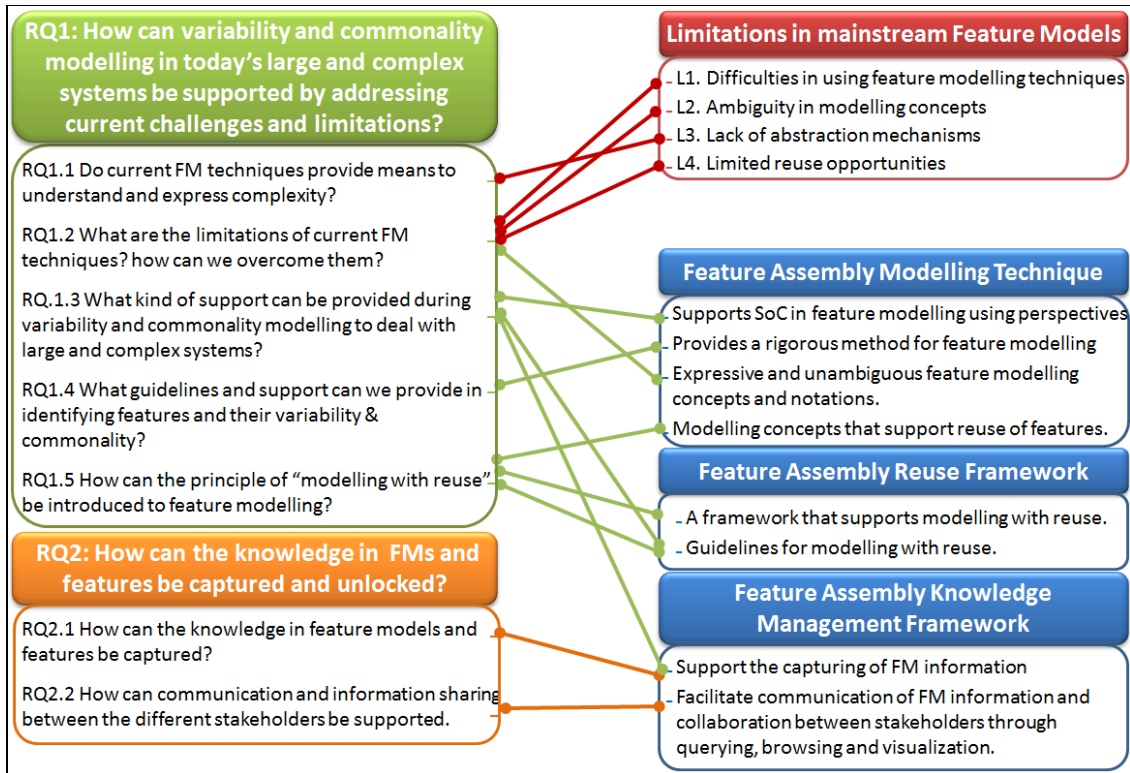


Figure 12.1: Overview of the work presented in this thesis in relation to our research questions.

Next, we have analysed these issues in order to identify a set of recommendations to overcome them. This was the base of our Feature Assembly Approach. Furthermore, we believe that the reusability supported by the variability offered by the concept of software product line is quite limited, as it should be possible to reuse the same feature with different variability specifications in different products. Furthermore, reuse should be supported at the modelling level; therefore we also aimed supporting feature reuse among different product lines (and products) at the level of the feature specification. In order to do so, there was a need to separate the definition of a feature from how it contributes to the variability. Additionally, there was a need to allow different participants involved in the software development process to share and collaborate their knowledge. It is important that this knowledge is readily available and that it supports the team's need to understand and analyse the complexity and gain a better understanding of existing variability opportunities. We defined the following requirements:

1. Support for a new feature modelling technique that satisfies the following requirements:
 - a. Provides abstraction mechanisms to deal with complex and large systems.
 - b. Provides a rigorous methodology for feature modelling.
 - c. Provides unambiguous modelling concepts with intuitive meaning. In particular, separates the feature from how it contributes to variability; it must be possible to reuse the same feature in different variability specifications.
2. Support efficient information processing and knowledge management of feature models; the following requirements should be supported:

- a. Allow users to share information about feature models, and allow them to query for information contained in feature models.
- b. Make it possible to support different abstraction mechanisms when viewing information about feature models.

The first step for defining the Feature Assembly approach was the development of the *Feature Assembly Modelling technique (FAM)*. FAM is a conceptual feature modelling technique that aimed to overcome the limitations of mainstream feature modelling techniques (identified by RQ1.1. and RQ1.2 as already mentioned). The main characteristics and the relations to our research questions are as follows:

- It uses *perspectives* as abstraction mechanisms, allowing features to be defined from different points of view, thus answering research question **RQ1.3**. A product line consists of one or more perspectives (e.g. Graphical User Interface Perspective, Functional Perspective, User Perspective). It is up to the modeller to select or define perspectives suitable for his product (line). In addition, we have provided guidelines to assist modellers during modelling, thus answering research question **RQ1.4**. We have also predefined a set of possible perspectives; at the same time this set is extensible and which perspectives are used is dependent on the domain of the product line. The concept of “feature” differs according to the perspective considered; therefore we have provided guidelines for defining features in each perspective.
- The modelling language provided uses simple feature modelling primitives, and take into account the need for models to be flexible enough to support evolution of the created models and the need to support reuse, thus answering research question **RQ1.2** and contributing to the answer for research question **RQ1.5**.

We only consider two types of features: *Feature* and *Abstract Feature*. A *Feature* represents a concrete logical or physical unit or characteristic of the system. An *Abstract Feature* is a feature that is not concrete; rather it is a generalization of more specific features (concrete or abstract ones). How the features are assembled together to compose the system is specified via *feature relations*. We have defined two types of feature relations: *composition relation* and *generalization/specification relation*. The *composition relation* is used to express the whole-part relation; i.e. a feature is composed of one or more fine-grained features. The composition can be *mandatory* or *optional*. The *generalization/specification relation* is used to represent is-a relations. In terms of variability, an abstract feature represents a variation point. Its available option features (i.e. specifications) represent variants. The number of option features allowed to be selected for a certain product is expressed via a *cardinality* constraint.

Furthermore, we define a set of *feature dependencies* that allow expressing dependencies between features. We allow feature dependencies to be expressed between features from a single perspective as well as between features from different perspectives. Feature dependencies between features from different perspectives glue the different perspectives together.

- Furthermore, we pointed out the need to link data variability with the variability of the system features. Therefore, we introduced the *Persistent* perspective for modelling data intensive variable applications. We showed how the variability in the application features triggers variability in the underlying data model and how this link can be maintained by using variability *annotations* in the data model.

We have demonstrated with an example how the Feature Assembly Modelling approach can be used. We showed the modelling process for a family of applications to create web-based interactive quizzes.

The next step for realizing Feature Assembly was proposing a concrete reuse mechanism in order to answer our research question **RQ1.5**; therefore, we defined the *Feature Assembly Reuse Framework*.

- The Feature Assembly Reuse Framework allows reusing features from a repository, called the *Feature Pool*. The Feature Pool is populated with features whenever the development for a new product takes place. Features are incrementally added to the feature pool, letting it act as a central storage of features. When a new product is required, the feature pool should be searched in order to find existing features that (partially) match the needs of this new product. In this way, allowing new products to be assembled from already existing features in addition to newly introduced ones.

Our final step for realizing Feature Assembly was to answer our third research question **RQ2**. We do so by providing a processable representation of the Feature Assembly models and the Feature Pool in order to allow users to interact with and share the information they hold. In order to do so, we adopted a knowledge-based approach and created the following artefacts:

- We created the *Feature Assembly Model Ontology* (FAM Ontology), which is a processable ontological model to represent the Feature Assembly models. The FAM Ontology acts as a formal documentation store for the information contained in the Feature Assembly models and it allows users to easily retrieve this information. We provided two approaches for interacting with the information, namely browsing the information (via a general purpose ontology browser) and querying the information (we showed examples using SPARQL). Additionally, we showed that applying these two approaches via a *dedicated Feature Assembly ontology browser* is both more intuitive and more user friendly. Furthermore, the FAM Ontology also allows using the power of Description Logic for enforcing the Feature Assembly models formulation rules. Furthermore, we have identified a set of rules that allow the ontology to isolate the set of features that cause some modelling errors: cycles in feature dependencies, inconsistencies between features dependencies, redundant feature dependencies, and cardinality errors.
- We have also created the *Feature Pool Ontology* which is the processable representation of the Feature Pool; the Feature Pool Ontology was actually extracted from the FAM Ontology.

We have also validated the Feature Assembly approach with a company. We presented our experience in actually applying the approach in a medium scale software company. This experience was fruitful in many ways. For this particular case, it confirmed our hypothesis about the importance of the variability modelling phase and the impact of a good design on future expansion of the software, and for reusing parts of it in other applications. Furthermore, it has also confirmed that even with small-scale software and a small team there is a need for managing and interacting with the feature model information. This experience has also helped us refine our approach and discover some limitations and interesting future work.

12.2 Contributions and Achievements

In this section, we summarize the major contribution and achievements of the work presented in this thesis. The contributions of our proposed solution are based on addressing the limitations and practical issues of current feature modelling techniques. While current research is devoted to automatic validation of feature models, we believe that there is still a need for improvement in the feature modelling technique. While doing so, we also proposed answers to some of the unaddressed issues in current variability modelling practice. The addressed issues stem from the need to manage and communicate the large amount of knowledge concerning the software's features, their commonality, and variability.

In the field of variability modelling, this thesis builds upon the analogy between assembling “parts” in industry and assembling “features” in software. Earlier, a similar analogy has been made between assembling “parts” in industry and assembling “code” (i.e. components) in software, however our approach is different in the sense that we introduce the assembling as early as possible, i.e. during design time. Furthermore, we argue that reuse is more effective if planned at domain analysis time. This allows making a design with reuse in mind and could significantly help in coming up with more modular (and therefore reusable) system. In addition, this has allowed us to combine “design for reuse” with “design with reuse”, which also reduce the design effort. To the best of our knowledge no work exists on proposing reuse of features or partial feature model. We hope that the work presented in this thesis sheds the light to the importance of supporting such reuse of features and partial feature models. On the one hand it allows for reusing previous knowledge of the domain. On the other hand, reuse at the domain analysis phase should strengthen the reuse opportunities at design and architectural phases.

This thesis emphasises the importance of the conceptual modelling of software variability and commonality information. In this thesis, we take the position “conceptual models are created by humans for humans”, sending the message that good feature models are a medium to convey knowledge on the variability, commonality, relations and dependencies of software features. We believe that the quality of the feature models should not be biased by the modeller engaging into the modelling process, rather the feature modelling language should be rigorous enough to support the modeller create unambiguous feature models. In the meantime the created models should be simple and intuitive enough for other stakeholders to understand. Therefore, unlike FODA and subsequent feature modelling techniques, the feature assembly modelling technique supports modellers in defining feature models that express complexity, are scalable, and unambiguous. Taking into account that modelling is a process which involves many stakeholders; this should also contribute to a more effective feature modelling process. Unlike current feature modelling techniques, we do not limit the modelling to a top down hierarchical modelling; rather we allow a combination of both top-down and bottom-up modelling for the sake of ease of modelling. We also use the concept of SoC to handle complexity and feature modelling of large systems through defining the concept of “Perspectives”. We do not restrict the modeller to a specific set of perspectives unlike for example FORM [Kang et al., 1998] which restricts the modeller to four categories for defining the features of the system.

As a contribution to the field of software modelling in general, the work presented contributes to satisfying the observed need of information sharing and the unlocking of information contained in software models. Software models are often stored as diagrams without proper means for querying them; they are often difficult to understand by non-technical people; and they may become very large. To overcome this, we have provided an integrated solution based on a processable information model. We show how this model can be used to create a visual and interactive feature model browser that can improve the collaboration

between different stakeholders in analysing, adding and retrieving information. Although there has been some works on visualizing feature models, interactive visualization and query support was missing.

The concrete contributions of this thesis can be divided into contributions to the *modelling* of software variability, and contributions to the *information management* of software variability. In the domain of modelling software variability we have the following contributions:

1. **The Feature Assembly Modelling language**, which provides only a few modelling concepts but with clear meaning. The language allows modelling features, their variability, their relations and their dependencies. This language overcomes the problems found in current mainstream variability languages.
2. **The introduction of the concept of “perspective”** as an abstraction mechanism during modelling for dealing with large (variable) software. Perspectives provide separation of concerns and ease the modelling as trying to deal with all aspects of software at the same moment is very difficult and will usually result in badly structured and large models.
3. **The separation of the feature from how it contributes to variability** allows defining feature assembly models that are easy to change and allows features to be reused in other designs.
4. **The Feature Assembly Reuse Framework**, which promotes modelling for reuse and modelling with reuse. This allows making reuse of previous feature models; features or even partial feature models can be shared between applications belonging to the same domain or to similar domains.
5. **The Feature Pool concept**, which enables reuse of features by storing and documenting them.

In the domain of managing information about software variability we have the following contributions:

1. **An ontology-based mechanism for representing, and validating variability information**, this includes the definition of the FAM Ontology which enables storing and interacting (i.e. searching and browsing) with Feature Assembly Models; using SWRL rules to detect modelling errors that may occur in the Feature Assembly models.
2. **The Feature Assembly Ontology Browser** which is a dedicated browser that allows visualizing, interacting with and searching information represented in Feature Assembly models.
3. **The Feature Pool Ontology**, which enables storing of reusable features, as well as searching for features or simply exploring the complete feature space.

12.3 Limitations

In this section, we discuss the boundaries and limitations of our work:

- **No Support for N-ary Feature Dependencies**
In this thesis we restrict ourselves to binary feature dependencies, as they are easier to understand and define (by modellers) than n-ary dependencies (as already mentioned in section 6.5.3). We had to decide whether the power of n-ary feature dependencies (which

will allow modellers to specify rather complex feature dependencies, as we proposed in [Abo Zaid et al., 2010]) outweighed the added complexity they introduce (e.g., checking their consistency). Furthermore, in most situations complex dependencies may be reduced to a set of binary feature dependencies. As we have not faced a case in which a binary dependency was not sufficient to model the required feature dependencies, we opt, for the sake of the modeller, for the simplicity of the binary dependencies.

- **Limitations in detecting Modelling Errors**

We concentrated on syntactical and common semantic errors. Yet there is no guarantee that we cover all possible inconsistencies and semantic errors that may occur. For example, the modeller might over-constrain the model such that no valid configuration can be found. Currently we do not check this, as it requires a constraint solver to validate that the model has feasible solutions. Furthermore, it could also be the case that the modeller over-constrains a certain feature such that it is no longer possible to select it in any valid configuration (i.e. it becomes a dead feature), we currently do not detect this error.

12.4 Future Work

The Feature Assembly Approach opens the way to new interesting future work. Some work concerns straightforward elaborations of the work presented or extensions to the current approach; others are more challenging and will require more research investigation. We list these different possibilities for future work:

- *Version Control for the Feature Assembly Approach*

Evaluating the Feature Assembly approach in an industrial setting has revealed the need for adding some type of *version control* for features. This is extremely relevant when using the Feature Pool, as overtime some features may need an update in their internal structure and therefore multiple versions of the same feature may exist in the pool. Therefore, there is a need to be able to track all different versions of a certain feature, in addition to which version of the feature is used in an application. Also, some features may not be relevant anymore in new products and therefore it should be possible to flag these features as deprecated.

- *Additional meta-data for Feature Assembly Models*

We have identified a basic set of meta-data information that needs to be associated with a certain feature. Yet more research is required in order to extend this set with *additional meta-data*. Enriching the features (and therefore the Feature Pool) with meta-data guarantees more efficient retrieval of information when searching for specific features. Part of this metadata may also be company specific, for example if a company associates a certain working scheme for their teams in order to facilitate the interoperability of information inside the company.

- *Linking features with code-artefacts*

Additionally, it would be interesting to explicitly link Feature Assembly Models to code artefacts. This will allow maintaining a link between the models and the corresponding code. This allows tracing the impact of changes in the model (e.g., for maintenance or evolution of the product) on the code. Also, this will allow realizing reusability at the architecture level. One possibility for this link between the features and the code could be via defining the appropriate meta-data associated to the feature that allows to indicate to which code artefact(s) this feature corresponds to. A one-to-one mapping between features

and code artefacts may not always be feasible and rather difficult to realize due to the many tangling concerns between parts of code, classes and components involved. Therefore, there may be a need to investigate the use of configuration languages as a kind of middle layer between the high-level feature models and the low-level implementation components. Configuration languages can abstract over low-level code composition, therefore, establishing a more feasible link between the code and the features. And therefore, providing support for the extended hypothesis of Feature Assembly approach which could be: “generating software by assembling features (from the Feature Pool) that make up a product”.

- *Enhanced reasoning on the FAM Ontology*

One of the merits of using Semantic Web technology for representing the Feature Assembly models and the information contained in the Feature Pool is the possibility to semantically process this ontology for retrieving new interesting information. For example, a “*similar to*” relation may be added for features that have the same set of feature dependencies, or have a number of common tags. A “*used together with*” relation may be defined to provide recommendations for reusing features when reusing a certain feature. More research is required to identify this set of “semantic” relations that may be of relevance for features/perspectives defined within the Feature Pool. The goal is to provide a better understanding of the hidden relations between features defined in the feature pool, and make the best use of these relations to improve reuse opportunities. Eventually they may also serve as design guidelines, this needs more investigation.

- *Improved Feature Pool Information Visualization*

A prototype for visually browsing Feature Assembly models is the FAM Ontology browser is presented in chapter 10, which is also suitable for browsing the Feature Pool. In addition to the current features of the FAM Ontology browser, users should be able to visually build queries that query the Feature Pool and visualize the query results. To deal with the large size of the pool, multiple visualizations should be supported. Additionally, users should be able to select from different abstraction levels, these abstraction levels could relate to their roles for example. More research is required on the adequate visualization techniques to use and the best user interactivity supported; some user validation is also required.

- *Feature Assembly Modelling Tool Support*

There is a need for a Feature Assembly Modelling tool that allows users to visually create Feature Assembly models, which are then stored in the FAM Ontology. We believe that from a usability point of view visually modelling and editing Feature Assembly models would be more appealing to users than adding this information via an ontology editing tool (e.g., Protégé as already indicated in chapter 10) or via a form-based method (as already indicated in chapter 11). One way to do so is via a diagram editor generator that makes use of meta-model-based language specifications. In that case, in order to create an editor for a specific diagram language, the editor developer has to provide two specifications: First, the abstract syntax of the diagram language in terms of its model, and secondly, the visual appearance of the diagram components (please refer to chapter 6.5 for the FAM model syntax and semantics).

The development of such a tool triggers some research questions related to the usability of the tool. As the number of features may become very large, as well as the number of relations between features, there is a need for good feature model management and visualization. In order to deal with the large size of the models, several views and abstraction levels should be possible. Furthermore, there is a need to select the best methodology to visualize perspectives, and how intra-perspective feature relations can be defined via the tool. Additionally a list of facilities to enhance the user experience with the tools should be provided, such as allowing information search in the generated models (i.e. graphs), allowing information projection, supporting feature comparison, etc. Therefore, more research will be needed to select the most appropriated visual notations, interaction techniques, supported abstraction mechanisms, etc. Additionally, usability experiments should be performed.

Furthermore, there is a need for a Feature Assembler tool that facilitates the actual assembly process of creating Feature Assembly models from already existing features in the Feature Pool in addition to the newly defined features. For this purpose, an interface to the Feature Pool should be defined to allow retrieving features from the Feature Pool and to add newly defined features to the Feature Pool.

- *Feature Assembly Configuration Tool Support*

Throughout this thesis our main concern was the conceptual modelling of the variable software. Nevertheless for a complete solution there is a need for allowing users to view the set of feasible products that a certain feature model defines. As already mentioned the Feature (Assembly) Models represent a Constraint Satisfaction Problem (CSP), therefore it is possible to use off-the-shelf constraint satisfaction solvers to automatically calculate the number of possible configurations, detect void features, and detect possible conflicts; as explained by Benavides et al. [2005].

Therefore it would be interesting to add configuration support to the Feature Assembly Modelling Tools. For example, by providing an option “Calculate Products” which communicates with the constraint solver, sends to it the encoded Feature Assembly model and retrieves back the set of feasible solutions (i.e. possible product configurations). Another alternative would be to make this link via the FAM Ontology, through a tool that encodes the information contained in the FAM Ontology to be read by the constraint solver, sends it the encoded model and retrieves back the set of feasible solutions to show to the user.

- More evaluation

Applying the Feature Assembly approach to more industrial cases will certainly help improving the technique. It will also help understand which of the above-mentioned future work will be most relevant for companies and should be given priorities. One interesting scenario to evaluate is the appropriateness of using the Feature Assembly approach for companies moving from many customized products to one variable product. There is a need to explore how the idea of the Feature Pool could help these companies in the *productization process*⁷⁰. For example, Feature Assembly Models could explicitly sketch the variability among features of the resulted customized products. These models can then be enriched with metadata to help the different involved stakeholders find the necessary

⁷⁰ Transforming from developing customer-specific software to product software is referred to as Productization [Artex et al., 2010]

information. Additionally the feature pool would act as a store for the product portfolio of the company.

Another interesting scenario is to use Moody's Method Evaluation Model [2003] to evaluate and compare the *semantic quality*⁷¹ and the *perceived semantic quality*⁷² of feature models and feature assembly models. This should allow to compare the two techniques based on both actual and perception-based properties for measuring the efficiency and effectiveness of a modelling method. Based on this, a set of rigorously defined empirical tests should be set up to provide proper evidence that the work proposed is pragmatically an improvement over existing approaches as well as semantically (we provided some theoretical evidence that the Feature Assembly modelling technique is semantically an improvement over current feature modelling techniques in section 6.6).

⁷¹ The *semantic quality* expresses the degree of correspondence between the information conveyed by a model and the domain that is modelled [Poels et al., 2005].

⁷² The *perceived semantic quality* measures perceptions of semantic quality as perceived by users [Poels et al., 2005], i.e the correspondence between the user interpretation (what a user thinks a model depicts) and the domain knowledge [Figl and Derntl, 2011].

List of References

- (1983). Software Technology for Adaptable, Reliable Systems (STARS) program strategy, ACM SIGSOFT Software Engineering Notes, Vol.8 issue.2 April, pp.56-108.
- (1996). DOD Software Reuse Initiative, ftp://ftp.cs.kuleuven.ac.be/pub/Ada-Belgium/ase/ase02_01/bookcase/se_sh/whyreuse/index.htm#xtocid222690 , last visit 11/7/2012
- Aamodt, A. (1995). Knowledge Acquisition and Learning from Experience - The Role of Case-Specific Knowledge, In Gheorge Tecuci and Yves Kodratoff (eds): Machine learning and knowledge acquisition; Integrated approaches, (Chapter 8), Academic Press, 1995, pp. 197-245.
- Abo Zaid, L., Kleinermann, F., De Troyer, O. (2009). Applying Semantic Web Technology to Feature Modeling. In: The 24th Annual ACM Symposium on Applied Computing, The Semantic Web and Applications (SWA) Track.
- Abo Zaid, L., Kleinermann, F., De Troyer, O. (2010 a). Feature Assembly: A New Feature Modeling Technique. In : 29th International Conference on Conceptual Modeling, Lecture Notes in Computer Science, Vol. 6412/2010, pp. 233-246
- Abo Zaid, L., Kleinermann, F., De Troyer, O. (2010 b). Feature Assembly Modelling: A New Technique for Modelling Variable Software", 5th International Conference on Software and Data Technologies Proceedings, Volume: 1, pp. pp: 29 - 35, Eds. José Cordeiro Maria Virvou Boris Shishkov, Publ. SciTePress, ISBN 978-989-8425-22-5, Athens, Greece
- Abo Zaid, L., De Troyer, O. (2011). Towards Modeling Data Variability in Software Product Lines, T. Halpin et al. (Eds.): BPMDS 2011 and EMMSAD 2011, LNBIP 81, pp. 453--467. Springer, Heidelberg (2011)
- Abo Zaid, L., Kleinermann, F., De Troyer, O. (2011). Feature Assembly Framework: towards scalable and reusable feature models, In Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems (VaMoS '11). ACM, New York, NY, USA, pp. 1-9
- Abo Zaid, L., De Troyer, O (2012). Modelling and managing variability with feature assembly: an experience report. In Proceedings of the Second Edition of the International Workshop on Experiences and Empirical Studies in Software Modelling (EESSMod '12). ACM, New York, NY, USA, DOI=10.1145/2424563.2424575
- Acher, M., Collet, P., Lahire, P., France, R. (2009). Composing Feature Models. In: Proceedings of 2nd International Conference on Software Language Engineering (SLE'09)
- Acher, M., Collet, Ph., Lahire, Ph., France, R. B. (2012). Separation of concerns in feature modeling: support and applications. In Proceedings of the 11th annual international conference on Aspect-oriented Software Development (AOSD '12). ACM, New York, NY, USA, pp.1-12. DOI=10.1145/2162049.2162051
- Ajila, S. A., and Kaba, A.B. (2008). Evolution support mechanisms for software product line process. J. Syst. Softw. 81, 10 (October 2008), 1784-1801. DOI=10.1016/j.jss.2007.12.797
- Alavi, M. and Leidner, D. E. (2001). Knowledge Management and Knowledge Management Systems: Conceptual Foundations and Research issues. MIS Quarterly, Vol. 25, No. 1, pp. 107-136.
- Alexander, I. F. , Maiden, N. (2004) . Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle. Wiley.
- Allemang, D, Hendler, J. A. (2008) Semantic web for the working ontologist - modeling in RDF, RDFS and OWL. Elsevier 2008: I-XVII, 1-330

- Anderson, J.R. (1996). ACT: A simple theory of complex cognition. *American Psychologist*, 51 (4), 355-365.
- Anquetil, N., Kulesza, U., Mitschke, R., Moreira, A., Royer, J-C., Rummler, A., Sousa, A. (2010). A model-driven traceability framework for software product lines. *Softw. Syst. Model.* 9, 4 (September 2010), 427-451. DOI=10.1007/s10270-009-0120-9
- Artz, P., Van De Weerd, I., and Brinkkemper, S. (2010). Productization: The process of transforming from customer-specific software development to product software development. In *Proceedings of ICSOB 2010*, pp. 90-102. DOI=10.1.1.166.7
- Asikainen, T. (2004). *Modelling Methods for Managing Variability of Configurable Software Product Families*. Licentiate thesis. Helsinki University of Technology, Department of Computer Science and Engineering
- Asikainen, T., Männistö, T., Soininen, T. (2006). A Unified Conceptual Foundation for Feature Modelling. In *Proceedings of the 10th International on Software Product Line Conference (SPLC '06)*. IEEE Computer Society, Washington, DC, USA, pp.31-40.
- Asikainen, T., Männistö, T., Soininen, T. (2007). Kumbang: A Domain Ontology for Modeling Variability in Software Product Families. *Advanced Engineering Informatics*, 21(1), pp. 23-40
- Aurum, A., Daneshgar, F., Ward, J. (2008). Investigating Knowledge Management practices in software development organisations – An Australian experience. In: *Information and Software Technology*. Vol. 50, pp. 511-533.
- Atkinson, C., Bayer, J., Muthig, D. (2000). Component-based product line development: the KobrA approach. In *Proceedings of the first conference on Software product lines : experience and research directions: experience and research directions*, Patrick Donohoe (Ed.). Kluwer Academic Publishers, Norwell, MA, USA, pp. 289-309.
- Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., Patel-Schneider, P. F. (2003). *The Description Logic Handbook: Theory, Implementation, and Applications* Cambridge University Press
- Babenko, L. P. (2003). Information Support of Reuse in UML-Based Software Engineering. *Cybernetics and Sys. Anal.* 39, 1 (January 2003), pp.65-70. DOI=10.1023/A:1023821025887 <http://dx.doi.org/10.1023/A:1023821025887>
- Bąk, K., Czarnecki, K., Wąsowski, A. (2010). Feature and meta-models in Clafer: mixed, specialized, and coupled. In *Proceedings of the Third international conference on Software language engineering (SLE'10)*, Brian Malloy, Steffen Staab, and Mark Van Den Brand (Eds.). Springer-Verlag, Berlin, Heidelberg, pp.102-122.
- Bartholdt, J., Oberhauser, R., Rytina, A. (2008). An Approach to Addressing Entity Model Variability within Software Product Lines?, *ICSEA 2008*. pp. 465-471
- Bartholdt, J., Oberhauser, R., Rytina, A. (2009). Addressing Data Model Variability and Data Integration within Software Product Lines *The International Journal On Advances in Software*, vol. 2, no. 1, pp. 86-102, ISSN 1942-2628.
- Batini, C., Lenzerini, M., Navathe, S. B. (1986). A comparative analysis of methodologies for database schema integration. *ACM Comput. Surv.* 18, 4 (December 1986), 323-364. DOI=10.1145/27633.27634
- Batista, V.A., Peixoto, D. C. C., Pádua, W., Pádua, C. I. P. S. (2012). Using UML Stereotypes to Support the Requirement Engineering: A Case Study. In *Proceedings of the 12th international conference on Computational Science and Its Applications - Volume Part IV (ICCSA'12)*, Beniamino Murgante, Osvaldo Gervasi, Sanjay Misra, Nadia Nedjah, and Ana C. Rocha (Eds.), Vol. Part IV. Springer-Verlag, Berlin, Heidelberg, pp.51-66. DOI=10.1007/978-3-642-31128-4_5
- Batory, D. (2005). Feature models, grammars, and propositional formulas. In: *Obbink, H., Pohl, K. (eds.) SPLC 2005*. LNCS, vol. 3714
- Behjati, R., Yue, T., Briand, L. C. (2012). A Modeling Approach to Support the Similarity-Based Reuse of Configuration Data. In *Proceedings of the 15th international conference on Model Driven Engineering Languages and Systems (MODELS'12)*, Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson (Eds.). Springer-Verlag, Berlin, Heidelberg, pp.497-513.

- Benavides, D., Trinidad, P., Ruiz-Cortés, A. (2005). Automated Reasoning on Feature Models. In: Proceedings of 17th Conference on Advanced Information Systems Engineering (CAiSE'05) ,
- Benavides, D., Segura, S., Antonio Ruiz-Cortés, A. (2010). Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.* 35, 6 (September 2010), 615-636. DOI=10.1016/j.is.2010.01.001
- Berners-Lee, T., Connolly, D. (2011). Notation3 (N3): A readable RDF syntax, <http://www.w3.org/TeamSubmission/n3/>, last visit 1/8/2012
- Berners-Lee, T., Hendler, J., Lassila, O. (2001). The Semantic Web: A New Form of Web Content that Is Meaningful to Computers Will Unleash a Revolution of New Possibilities, *Scientific American Journal*, May 2001, pp. 28–37
- Bjørnson, F. O., Dingsøy, T. (2008). Knowledge management in software engineering: A systematic review of studied concepts, findings and research methods used. *Inf. Softw. Technol.* 50, 11 (October 2008), pp. 1055-1068. DOI=10.1016/j.infsof.2008.03.006
- Bontemps, Y., Heymans, P., Schobbens, P.-Y., Trigaux, J.-C. (2004). Semantics of FODA Feature Diagrams. In: Workshop on Software Variability Management for Product Derivation Towards Tool Support, (T. Männistö & J. Bosch, Eds.)System, (Theorem 2), pp. 48-58
- Borgida, A. (2007). How knowledge representation meets software engineering (and often databases). *Automated Software Engg.* 14, 4, pp. 443-464. DOI=10.1007/s10515-007-0018-0
- Bosch J. (2009). From software product lines to software ecosystems, in: Proc. of the 13th International Software Product Line Conference (SPLC 2009), Software Engineering Institute, CarnegieMellon, San Francisco, CA, USA, 2009, pp. 111– 119.
- Bosch, J. (2000). Design and Use of Software Architectures: : Adopting and Evolving a Product-Line Approach., Addison-Wesley.
- Bosch, J. (2005). Software Product Families in Nokia. In: 9th International Conference SPLC 2005 (2005).
- Bosch, J. (2010). Toward compositional software product lines, *IEEE Software* 27 (3) pp.29–34.
- Bray, I. (2002). An introduction to requirements engineering, Pearson Addison Wesley
- Buhne, S., Lauenroth, K., Pohl, K. (2005). Modelling Requirements Variability across Product Lines. In Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE '05). IEEE Computer Society, Washington, DC, USA, 41-52. DOI=10.1109/RE.2005.45 <http://dx.doi.org/10.1109/RE.2005.45>
- Bylander, T. , Chandrasekaran, B. (1988). Generic Tasks in knowledge-based reasoning. the right level of abstraction for knowledge acquisition. In: Knowledge Acquisition for Knowledge Based systems, pp. 65-77, London: Academic Press
- Cawley, C., Nestor, D., Preußner, A., Botterweck, G., Thiel, S. (2008). Interactive Visualisation to Support Product Configuration in Software Product Lines, Proceedings of the 2nd International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS 2008), Essen, Germany, January 16-18, pp. 7-16. ISSN 1860-2770
- Cawley, C., Botterweck, G., Healy, P., bin Abid, S., Thiel, St. (2009). A 3D Visualisation to Enhance Cognition in Software Product Line Engineering. In Proceedings of the 5th International Symposium on Advances in Visual Computing (ISVC '09), Part II ISVC (2), pp.857-868
- Cawley, C., Healy, P., Botterweck, G., Thiel, St. (2010). Research Tool to Support Feature Configuration in Software Product Lines, In proceedings of VaMoS 2010, pp. 179-182
- Chen, L., Babar, M.A. (2010) Variability Management in Software Product Lines: An Investigation of Contemporary Industrial Challenges. In Proceedings of the 14th international conference on Software product lines: going beyond (SPLC'10), Jan Bosch and Jaejoon Lee (Eds.). Springer-Verlag, Berlin, Heidelberg, pp.166-180.

- Chen, L., Babar, M.A., Ali, N. (2009). Variability management in software product lines: a systematic review. In Proceedings of the 13th International Software Product Line Conference (SPLC '09). Carnegie Mellon University, Pittsburgh, PA, USA, pp.81-90.
- Chen, P. P. (1976). The Entity-Relationship Model - Toward a Unified View of Data, *ACM Trans, Database Syst.* 1(1): 9-36
- Classen, A., Heymans, P., and Schobbens, P-Y. (2008). What's in a feature: a requirements engineering perspective. In Proceedings of the Theory and practice of software, 11th international conference on Fundamental approaches to software engineering (FASE'08/ETAPS'08), Fiadeiro, J. and Inverardi, P. (Eds.). Springer-Verlag, Berlin, Heidelberg, pp.16-30.
- Classen, A., Boucher, Q. and Heymans, P. (2011). A Text-based Approach to Feature Modelling: Syntax and Semantics of TVL. In *Science of Computer Programming, Special Issue on Software Evolution, Adaptability and Variability*, 76 (12): pp. 1130-1143
- Clauss, M. (2001). Generic Modeling using UML extensions for variability. In *Workshop on Domain-specific Visual Languages, OOPSLA 2001*, pp. 11-18.
- Clements, P. C., Northrop, L. M. (2002). A Software Product Line Case Study", Technical Report CMU/SEI-2002-TR-038
- Codenie, W., González-Deleito, N., Deleu, J., Blagojevic, V., Kuvaja, P., Similä, P. (2009). A Model for Trading off Flexibility and Variability in Software Intensive Product Development. In proceeding of: Third International Workshop on Variability Modelling of Software-Intensive Systems, Seville, Spain, January 28-30, pp. 61-70.
- Connolly, T. M. , Begg, C. E. (2009). *Database Systems: A Practical Approach to Design, Implementation and Management*. Addison-Wesley, ISBN-10: 0321210255
- Creps, R.E. , Simos, M. A. (1992). The STARS Conceptual Framework for Reuse Processes. In Proceedings of the Fifth Annual Workshop on Software Reuse. Volume: 22091, Issue: 703, Citeseer
- Crnkovic, I., Chaudron, M., Larsson, S. (2006). Component-Based Development Process and Component Lifecycle, In: Proceedings of International Conference on Software Engineering Advances (ICSEA'06), pp.44
- Czarnecki, K., Eisenecker, U.W. (2000). *Generative Programming: Methods, Tools, and Applications*. Addison Wesley
- Czarnecki, K., Helsen, S., Eisenecker, U. W. (2004). Staged Configuration Using Feature Models. In proceedings of SPLC 2004. pp. 266-283
- Czarnecki, K., Kim, C. H. P. (2005). Cardinality-Based Feature Modeling and Constraints: A Progress Report. In: International Workshop on Software Factories at OOPSLA'05, San Diego, California, USA, ACM, 2005
- Czarnecki, K., Kim, C. H. P., Kalleberg, K.T. (2006). Feature models are views on ontologies. In Proceedings of the 10th International on Software Product Line Conference (SPLC '06). IEEE Computer Society, Washington, DC, USA, pp.41-51.
- Czarnecki, K., Gruenbacher, P., Rabiser, R., Schmid, K., Wasowski, A. (2012). Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In Proceedings of Variability Modelling of Software-intensive Systems (VaMoS), Leipzig, Germany, ACM, New York, NY, USA, pp.173-182. DOI=10.1145/2110147.2110167.
- David Beckett, Tim Berners-Lee, Turtle - Terse RDF Triple Language, <http://www.w3.org/TeamSubmission/turtle/>, 2011, last visit 1/8/2011
- Davis, R., Shrobe, H., Szolovits, P. (1993). What is a Knowledge Representation? *AI Magazine*, 14(1):17-33
- Della Valle, E., Ceri, S. (2011). Querying the Semantic Web: SPARQL. In: *Handbook of Semantic Web Technologies* pp. 299-363, Springer-Verlag Berlin Heidelberg
- Dhungana, D., Grünbacher, P., Rabiser, R., Neumayer, T. (2010). Structuring the modeling space and supporting evolution in software product line engineering, In: *Journal of Systems and Software* 83(7), pp.1108-1122

List of References

- Dieste, O., Juristo, N., Moreno, A. M., Pazos, J., Sierra, A. (2002). *Conceptual Modeling in Software Eng. and Knowledge Eng.: Concepts, Techniques and Trends*, In: *Handbook of Software Engineering and Knowledge Engineering*, Publisher: World Scientific.
- Doran, P., Tamma, V., Iannone, L. (2007). *Ontology Module Extraction for Ontology Reuse: An Ontology Engineering Perspective*. In *Proceedings of the ACM CIKM International Conference on Information and Knowledge Management*. November 6-9, 2007. Lisbon, Portugal
- Dustdar, S., Schreiner, W. (2005). *A survey on web services composition*, *International Journal of Web and Grid Services*, vol. 1, No. 1, pp. 1-30
- Eén, N., Sörensson, N. (2003). *An extensible SAT solver*. In: *6th International Conference on Theory and Applications of Satisfiability Testing*, LNCS 2919, pp. 502-518
- El Dammagh, M., De Troyer, O. (2011). *Feature modeling tools: evaluation and lessons learned*. In *Proceedings of the 30th international conference on Advances in conceptual modeling: recent developments and new directions (ER'11)*, Olga De Troyer, Claudia Bauzer Medeiros, Roland Billen, Pierre Hallot, and Alkis Simitsis (Eds.). Springer-Verlag, Berlin, Heidelberg, pp.120-129.
- Eriksson, M., Börstler, J., Borg, K. (2005). *The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations*. In: *Proceedings of the 9th International Conference on Software Product Lines (SPLC'05)*, LNCS, Vol. 3714, pp. 33-44, Springer-Verlag.
- Erwig, M. (1998). *Abstract Syntax and Semantics of Visual Languages*, *Journal of Visual Languages & Computing*, Volume 9, Issue 5, October 1998, Pages 461-483, ISSN 1045-926X, 10.1006/jvlc.1998.0098
- Fan, S. and Zhang, N. (2006). *Feature model based on description logics*. In: *Knowledge-Based Intelligent Information and Engineering Systems*, *Lecture Notes in Computer Sciences* Vol. 4252/2006, pp. 1144-1151, DOI: 10.1007/11893004_145
- Ferreira, N., Machado, R. J., Gasevic, D. (2009). *An Ontology-Based Approach to Model-Driven Software Product Lines*. In: *Proceedings of the 2009 Fourth International Conference on Software Engineering Advances (ICSEA '09)*. IEEE Computer Society, Washington, DC, USA, pp. 559-564. DOI=10.1109/ICSEA.2009.88
- Figl, K. and Derntl, M. (2011). *The impact of perceived cognitive effectiveness on perceived usefulness of visual conceptual modeling languages*. In *Proceedings of the 30th international conference on Conceptual modeling (ER'11)*, Manfred A. Jeusfeld, Lois Delcambre, and Tok Wang Ling (Eds.), pp.78-91. Springer-Verlag, Berlin, Heidelberg.
- Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., Goedicke, M. (1992). *Viewpoints: A Framework for Integrating Multiple Perspectives in System Development*, *Intl. J. of Software Engineering and Knowledge Engineering* 2(1), pp. 31-57
- Forbus, K.D., De Kleer, J. (1993). *Building Problem Solvers*. MIT Press
- Frakes, W. B., Díaz, R. P., Fox, C. J. (1998). *DARE: Domain Analysis and Reuse Environment*. In: *Ann. Software Eng.* 5: pp. 125-141
- Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley. ISBN-10: 0201633612
- Garcia-Molina, H., Ullman, J. D., Widom, J. (2008). *Database Systems: The Complete Book (2 ed.)*. Prentice Hall Press, Upper Saddle River, NJ, USA.
- Gašević, D., Devedžić, V. (2006). *Petri net ontology*, In: *Knowledge Based Systems*, Vol. 19, No. 4, 2006, pp. 220-234
- Gene Ontology, <http://www.geneontology.org/>, last visit 1/8/2011
- Gomaa, H. (2005). *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*, Addison-Wesley

- Graham, T.C. N. (1996). Viewpoints Supporting the Development of Interactive Software. In: Proceedings of Viewpoints 96: International Workshop on Multiple Perspectives in Software Development, ACM Press, San Francisco, USA, pp. 263-267
- Grenon, P. (2003). BFO in a nutshell: A bi-categorical axiomatization of BFO and comparison with DOLCE. Technical Report 06/2003, IFOMIS, University of Leipzig.
- Griss, M. L., Favaro, J., d'Alessandro, M. (1998). Integrating Feature Modeling with the RSEB, Proc. Fifth International Conference on Software Reuse, pp. 76-85, Victoria, BC, Canada
- Griss, M. L., (2000). Implementing Product-Line Features with Component Reuse. In Proceedings of the 6th International Conference on Software Reuse: Advances in Software Reusability (ICSR-6), William B. Frakes (Ed.). Springer-Verlag, London, UK, pp.137-152.
- Grönniger, H., Ringert, J. O., Rumpe, B. (2009). System Model-Based Definition of Modeling Language Semantics. In Proceedings of the Joint 11th IFIP WG 6.1 International Conference FMOODS '09 and 29th IFIP WG 6.1 International Conference FORTE '09 on Formal Techniques for Distributed Systems (FMOODS '09/FORTE '09). Springer-Verlag, Berlin, Heidelberg, pp.152-166.
- Gruber, T. R. (1993). Toward principles for the design of ontologies used for knowledge sharing. Presented at the Padua workshop on Formal Ontology, March 1993, later published in International Journal of Human-Computer Studies, Vol. 43, Issues 4-5, pp. 907-928
- Gruber, T. (2008). Ontology, Encyclopedia of Database Systems, Ling Liu and M. Tamer Özsu (Eds.), Springer-Verlag.
- Günther, S., Sunkle, S. (2012). rbFeatures: Feature-oriented programming with Ruby. Sci. Comput. Program. 77(3): pp.152-173
- Haarslev, V., Möller, R. (2001). RACER system description. In: Proceedings of the First International Joint Conference on Automated Reasoning (IJCAR 2001), Siena. Lecture Notes in Artificial Intelligence, vol. 2083, pp. 701–705. Springer, Berlin
- Haarslev, V., Möller, R. (2003). Racer: A Core Inference Engine for the Semantic Web. In Proceedings of the 2nd International Workshop on Evaluation of Ontology-based Tools (EON2003), pp. 27-36
- Halpin, T. A., and Bloesch, A. C. (1999). Data Modeling in UML and ORM: A Comparison. J. Database Manag. 10(4): pp. 4-13
- Halpin, T. , Morgan, T. (2008). Information Modeling and Relational Databases, Second Edition (ISBN: 978-0-12-373568-3), Morgan Kaufmann, 2008
- Harel, D., Rumpe, B. (2004). Meaningful Modeling: What's the Semantics of "Semantics"? In: Computer 37(10), pp. 64–72. DOI=10.1109/MC.2004.172
- Hartmann, H., and Trew, T. (2008). Using Feature Diagrams with Context Variability to Model Multiple Product Lines for Software Supply Chains. In Proceedings of the 2008 12th International Software Product Line Conference (SPLC '08). IEEE Computer Society, Washington, DC, USA, 12-21. DOI=10.1109/SPLC.2008.15
- Hayes-Roth., F., (1985). Rule-Based Systems. In: Commun. ACM, pp. 921-932
- Hebeler, J., Fisher, M., Blace, R., Perez-Lopez, A. (2009). Semantic Web Programming, Wiley Publishing, ISBN: 978-0-470-41801-7
- Heidenreich, F., and Wende, C. (2007). Bridging the Gap Between Features and Models. In Proceedings of the Second Workshop on Aspect-Oriented Product Line Engineering (AOPLE'07) co-located with the International Conference on Generative Programming and Component Engineering (GPCE'07), Salzburg, Austria, October 2007.
- Heidenreich, F., Kopcsek, J., Wende, and C. (2008). FeatureMapper: Mapping Features to Models. In Companion Proceedings of the 30th International Conference on Software Engineering (ICSE'08), Leipzig, Germany, May 2008.

- Heidenreich, F., Şavga, I. Wende, C. (2008). On Controlled Visualisations in Software Product Line Engineering. In Proceedings of the 2nd International Workshop on Visualisation in Software Product Line Engineering (ViSPLE 2008), collocated with the 12th International Software Product Line Conference (SPLC 2008), Limerick, Ireland, September 2008.
- Heineman, G.T., Councill, G.T. (2001). Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN 0-201-70485-4
- Hendler, J., and van Harmelen, F. (2008). The Semantic Web: Webizing Knowledge Representation. In: Foundations of Artificial Intelligence, vol. 3, Handbook of Knowledge Representation, pp. 821-839
- Hevner, A., and Chatterjee, S. (2010). Design Research in Information Systems, Springer Publishing Company, Incorporated. ISBN:1441956522 9781441956521
- Heymans, P., Boucher Q., Classen A., Bourdoux, A., Demonceau, L. (2012). A code tagging approach to software product line development - An application to satellite communication libraries. STTT 14(5): pp.553-566
- Holl, G., Grünbacher, P., Rabiser, R. (2012) A systematic review and an expert survey on capabilities supporting multi product lines. Information & Software Technology 54(8): pp.828-852
- Horrocks, I., Patel-Schneider, P. F. (2010). KR and Reasoning on the Semantic Web: OWL. In: Handbook of Semantic Web Technologies. chapter 9. Springer
- Horrocks, I. Sattler, U., Tobies, S. (1999). Practical Reasoning for Expressive Description Logics. In Ganzinger, H.; McAllester, D.; and Voronkov, A., eds., Proc. Of LPAR-6, vol. 1705 of LNAI, pp. 61–180. Springer.
- Horrocks, I., Kutz, O., Sattler, U. (2006). The Even More Irresistible SROIQ. In: Proceedings of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2006), pp.57-67. AAAI Press
- Horrocks, I., Patel-Schneider, P. F., Boley, H., Tabet, S., Grosz, B., Dean, M. (2004). SWRL: A Semantic Web Rule Language Combining OWL and RuleML , <http://www.w3.org/Submission/SWRL>, last visit 1/3/2011
- Horrocks, I., Patel-Schneider, P. F., van Harmelen, F. (2003). From SHIQ and RDF to OWL: The Making of a Web Ontology Language. In: Journal of Web Semantics, vol 1(1): pp.7-26
- Hubaux, A., Boucher, Q., Hartmann, H., Michel, R., Heymans, P. (2010 a). Evaluating a textual feature modelling language: four industrial case studies. In Proceedings of the Third international conference on Software language engineering (SLE'10), Brian Malloy, Steffen Staab, and Mark Van Den Brand (Eds.). Springer-Verlag, Berlin, Heidelberg, pp. 337-356.
- Hubaux, A., Classen, A., Mendonca, M., Heymans, P. (2010 b). A Preliminary Review on the Application of Feature Diagrams in Practice. In: Proceedings of the Fourth Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10), pp. 53-59
- Hubaux, A., Heymans, P., Schobbens, P-Y., Deridder, D. (2011). Supporting multiple perspectives in feature-based configuration. Software and Systems Modeling (SoSyM), pp.1-23, DOI 10.1007/s10270-011-0220-1
- Jacobson, I., Griss, M., Jonsson, P. (1997). Software Reuse: Architecture, Process and Organization for Business Success, Addison- Wesley-Longman
- Janota, M., Kiniry, J. (2007). Reasoning about Feature Models in Higher-Order Logic. In: Proceedings of 11th International Software Product Lines Conference (SPLC 2007).
- Jaring, M., Krikhaar, R. L., Bosch, J. (2004). Representing variability in a family of MRI scanners, In: Software—Practice & Experience, vol. 34 , Issue 1 , pp. 69 - 100
- Jarrar, M., Demey, J., Meersman, R. (2003). On Using Conceptual Data Modeling for Ontology Engineering, Lecture Notes in Computer Science, Vol. 2800, pp. 185-207

- Jekjantuk, N., Pan, J.Z., Qu, Y. (2011). Diagnosis of Software Models with Multiple Levels of Abstraction Using Ontological Metamodeling. In Proc. of the 35th IEEE Annual Computer Software and Applications Conference (COMPSAC 2011).
- Johansen, M. F., Fleurey, F., Acher, M., Collet, P., Lahire, P. (2010). Exploring the Synergies Between Feature Models and Ontologies. In Proceedings of the 14th International Software Product Line Conference. Volume 2 - Workshops, Industrial track, Doctoral symposium, Demonstrations and Tools. vol 2, pp. 163-171. Lancaster University, September 2010
- Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A. (1990). Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie-Mellon University
- Kang, K., Kim, S., Lee, J., Kim, K., Shin, E., and Huh, M. (1998). FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. In: J. Annals of Software Engineering. vol. 5, pp. 143-168.
- Kang, K.C., Lee, J., Donohoe, P. (2002). Feature-Oriented Product Line Engineering, IEEE Software, vol. 19, no. 4, pp. 58-65, July/Aug. doi:10.1109/MS.2002.1020288
- Keet, C.M. (2007). Mapping the Object-Role Modeling language ORM2 into Description Logic language DL^Rifd. KRDB Research Centre Technical Report KRDB07-2, Faculty of Computer Science, Free University of Bozen-Bolzano, Italy.
- Keet, C.M. (2008). A formal comparison of conceptual data modeling languages. 13th International Workshop on Exploring Modeling Methods in Systems Analysis and Design (EMMSAD'08). Montpellier, France, 16-17 June 2008. CEUR-WS Vol-337, pp25-39
- Kendal, S., and Creen, M. (2007). An Introduction to Knowledge Engineering, ISBN 13: 978-1-84628-475-5, Springer-Verlag
- Kersten, G. E., Kersten, M., Rakowski, W. M. (2002). Software and Culture: Beyond the Internationalization of the Interface. In JGIM. Vol. 10(4). pp. 86-101
- Kersten, G. E., Matwin, S., Noronha, S. J., Kersten, M. (2000). The Software for Cultures and the Cultures in Software. ECIS 2000, pp. 509-514
- Kimiz, D. (2005). Knowledge Management in Theory and Practice. Elsevier Butterworth-Heinemann
- Kolovos, D.S., Paige, R.F., Polack, F.A.C. (2006). Merging Models with the Epsilon Merging Language (EML), In: Proceedings of ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006).
- Korherr, B., and List, B. (2007). A UML 2 Profile for Variability Models and their Dependency to Business Processes. DEXA Workshops 2007: pp 829-834, ISBN: 0769529321, DOI: 10.1109/DEXA.2007.96
- Kossmann, M., Wong, R., Odeh, M., Gillies, A. (2008). Ontology-driven requirements engineering: building the OntoREM meta model. 3rd International Conference on Information and Communication Technologies: From Theory to Applications, 2008. ICTTA 2008. pp. 1-6.
- Kotiadis, K., and Robinson, S. (2008). Conceptual modelling: Knowledge acquisition and model abstraction. In: Proceedings of Winter Simulation Conference, pp. 951-958
- Kulak, D., and Guiney, E. (2003). Use Cases: Requirements in Context, Second Edition, Addison-Wesley Professional
- Lee, K., Kang, K. C., Lee, J. (2002). Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In: Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools (ICSR-7), Cristina Gacek (Ed.). Springer-Verlag, London, UK, UK, pp. 62-77.
- Lee, S.-B., Kim, J.-W., Song, C.-Y., Baik, D.-K. (2007). An Approach to Analyzing Commonality and Variability of Features using Ontology in a Software Product Line Engineering. SERA 2007. pp: 727-734

List of References

- Leenen, W., Vlaanderen, K., van de Weerd, I., Brinkkemper, S. (2012). Transforming to Product Software: The Evolution of Software Product Management Processes during the Stages of Productization. In Proceedings of ICSOB 2012: 40-54
- Lim, E. H. Y., James, L. N. K., Raymond S.T., L. (2011). Knowledge Seeker - Ontology Modelling for Information Search and Management A Compendium, Series: Intelligent Systems Reference Library, vol. 8, Springer-Verlag.
- Lopez-Herrejon, R.E., Batory, D. (2001). A Standard Problem for Evaluating Product-Line Methodologies. In: Bosch, J. (ed.) GCSE 2001. LNCS, vol. 2186, pp. 9–13
- Maccari, A., and Heie, A. (2005). Managing infinite variability in mobile terminal software. In: Research Articles. Softw. Pract. Exper. vol. 35, 6 (May 2005), pp. 513-537. DOI=10.1002/spe.v35:6
<http://dx.doi.org/10.1002/spe.v35:6>
- MacGregor, J. (2002). Bosch Experience Report, <http://www.conipf.org/download/BoschExperienceReport.pdf>, last visit 1/3/2011
- Mannion, M., Savolainen, J., Asikainen, T. (2009). Viewpoint-Oriented Variability Modeling, In: Proceedings of International Computer Software and Applications Conference (COMPSAC'09), pp. 67–72
- March, S., Smith, G. (1995). Design and natural science research on information technology. *Decis. Support Syst.* 15, 4 (December 1995), pp. 251-266. DOI=10.1016/0167-9236(94)00041-2
- Menéndez, V. H., and Prieto, M.E. (2008). A Learning Object Composition Model, In: Proceedings of UNISCON 2008, pp. 469-474
- Mohan, K., and Ramesh, B. (2003). Ontology-Based Support for Variability Management in Product and Service Familie. In: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 3 - Volume 3 (HICSS '03), vol. 3. IEEE Computer Society, Washington, DC, USA
- Moody, D. L. (2010). The "physics" of notations: a scientific approach to designing visual notations in software engineering. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2 (ICSE '10), Vol. 2. ACM, New York, NY, USA, 485-486. DOI=10.1145/1810295.1810442
- Moody, D. L. (2003). The method evaluation model: a theoretical model for validating information systems design methods. In proceedings of ECIS 2003, pp. 1327-1336
- Motik, B., Shearer, R., Horrocks, I. (2007) Optimized reasoning in description logics using hypertableaux. In: Proceedings of the 21st International Conference on Automated Deduction (CADE-21), Breman. Lecture Notes in Artificial Intelligence, vol. 4603, pp. 67–83. Springer, Berlin
- Mylopoulos, J. (2001). Conceptual Modeling for Knowledge Management: A Tutorial, In proceedings 9th IFIP 2.6 working conference on database semantics.
- N triples, <http://www.w3.org/TR/rdf-testcases/#ntriples>, 2004, last visit 1/3/ 2011
- Navathe, B. S., and Schkolnick, M. (1978). View representation in logical database design. In Proceedings of the 1978 ACM SIGMOD international conference on management of data (SIGMOD '78). ACM, New York, NY, USA, 144-156. DOI=10.1145/509252.509286
- Neighbors, J. (1984). The Draco Approach to Construction Software from Reusable Components, In: IEEE Transactions on Software Engineering SE-10, vol. 5, pp. 564-573
- Nestor, D., Thiel, S., Botterweck, G., Cawley, C., Healy, P. (2008). Applying visualisation techniques in software product lines. SOFTVIS 2008: pp.175-184
- Niemelä, I., and Simons, P. (1997). Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In: Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning, vol. 1265 of Lecture Notes in Artificial Intelligence, pp. 420-429

- Northrop, L., Feiler, P., Gabriel, R.P., Goodenough, J., Linger, R., Longstaff, T., Kazman, R., Klein, M., Schmidt, D., Sullivan K., Wallnau, K. (2006) Ultra-Large-Scale Systems – The Software Challenge of the Future: Software Engineering Institute, Carnegie Mellon, June 2006.
- Noy, N. F., McGuinness, D. L. (2001). *Ontology Development 101: A Guide to Creating Your First Ontology*. Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880
- Nuseibeh, B., Easterbrook, S., Russo, A. (2001) Making inconsistency respectable in software development. In *Journal of Systems and Software*, Volume 58, Issue 2, 1 September 2001, Pages 171-180, ISSN 0164-1212, 10.1016/S0164-1212(01)00036-X.
- Nuseibeh, B., Kramer, J., Finkelstein, A.C.W. (1994). A framework for expressing the relationships between multiple views in requirements specification. In *Trans. Software Eng.*, 20 (10) . pp. 760–773
- Nuseibeh, B., Kramer, J., Finkelstein, A. (2003). *ViewPoints: Meaningful Relationships Are Difficult!*. In: *Proceedings of International Conference on Software Engineering (ICSE'03)*
- Nyström, D., Tesanovic, A., Nolin, M., Norström, C., Hansson, J. (2004) COMET: A Component-Based Real-Time Database for Automotive Systems. In *Proceedings of the Workshop on Software Engineering for Automotive Systems at 26th International Conference on Software engineering (ICSE'04)*, IEEE Computer Society Press.
- Oberle, D., Lamparter, S., Grimm, S., Vrandečić, D., Staab, S., Gangemi, A. (2006). Towards ontologies for formalizing modularization and communication in large software systems. *Appl. Ontol.* 1, 2, pp.163-202.
- Object Constraint Language (OCL), <http://www.omg.org/technology/documents/formal/ocl.htm>
- Ossher, H., Tarr, P. (2001). Using Multidimensional Separation of Concerns to (re)shape Evolving Software. In: *Communications of the ACM*, vol. 44, pp. 43-49
- OWL 2 Web Ontology Language Document Overview <http://www.w3.org/TR/owl2-overview/>, 2009, last visit 1/3/2011
- OWL Web Ontology Language Overview, <http://www.w3.org/TR/owl-features/>, 2004, last visit 1/3/ 2011
- Pahl, C.: *Ontology Transformation and Reasoning for Model-Driven Architecture*. OTM Conferences (2) 2005: 1170-1187
- Palmer, S. R., and Felsing, J. M. (2001) . *A Practical Guide to Feature-Driven Development*, Prentice Hall, ISBN:0130676152
- Patel-Schneider, P. F., and Horrocks, I. (2004) OWL Web Ontology Language, Semantics and Abstract Syntax, http://www.w3.org/TR/2004/REC-owl-semantic-20040210/direct.html#owl_allValuesFrom_semantics
- Peffers, K., Tuunanen, T., Rothenberger, M. A., Chatterjee, S. (2008). A Design Science Research Methodology for Information Systems Research. *J. Manage. Inf. Syst.* 24, 3, pp. 45-77. DOI=10.2753/MIS0742-1222240302
- Peng, X., Zhao, W., Xue, Y., Wu, Y. (2006). Ontology-Based Feature Modeling and Application-Oriented Tailoring. In: *ICSR 2006*, pp. 87-100
- Pettersson, U., Jarzabek, S., (2005). Industrial Experience with Building a Web Portal Product Line using a Lightweight, Reactive Approach. In: *Proceedings of ESEC-FSE'05, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM
- Pizza ontology, <http://www.co-ode.org/ontologies/pizza/2007/02/12/>, last visit 1/3/ 2011
- Pohl, K., Böckle, G., van der Linden, F. (2005). *Software Product Line Engineering: Foundations, Principles, and Techniques*, Springer, ISBN 10-3-540-24372-0.
- Poels, G., Maes, A., Gailly, F., Paemeleire, R. (2005). Measuring the perceived semantic quality of information models. In *Proceedings of the 24th international conference on Perspectives in Conceptual Modeling (ER'05)*, Jacky

- Akoka, Stephen W. Liddle, Il-Yeol Song, Michela Bertolotto, and Isabelle Comyn-Wattiau (Eds.). Springer-Verlag, Berlin, Heidelberg, pp.376-385.
- Preece, A. Flett, A. Sleeman, D. Curry, D. Meany, N. Perry, P. (2001). Better Knowledge Management through Knowledge Engineering. *IEEE Intelligent Systems* 16(1), pp 36-43
- Puttemans, T. (2011). Querying and Exploring a Feature Pool, MSc. Thesis, Vrije Universiteit Brussel
- Rabiser, R., Grünbacher P., Dhungana D. (2010) Requirements for product derivation support: Results from a systematic literature review and an expert survey. *Information and Software Technology*, Volume 52, Issue 3, March 2010, pp. 324-346, ISSN 0950-5849, 10.1016/j.infsof.2009.11.001
- RDF Primer, <http://www.w3.org/TR/rdf-primer/>, 2004 , last visit 1/3/ 2011
- RDF/XML , <http://www.w3.org/TR/rdf-syntax-grammar/>, 2004, last visit1/3/ 2011
- Rector, A., Napoli, A., Stamou, G., Stoilos, G., Wolger, H., Pan, J. , D'Aquin, M., Spaccapietra, S., Tzouvaras, V. (2005). Report on modularization of ontologies. Technical report, Knowledge Web Deliverable, D2.1.3.1
- Reiser, M.-O. and Weber, M.(2006). Managing Highly Complex Product Families with Multi-Level Feature Trees. In *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE '06)*. IEEE Computer Society, Washington, DC, USA, pp. 146-155.
- Riebisch, M., Böllert, K., Streitferdt, D., Philippow, I. (2002). Extending Feature Diagrams with UML Multiplicities. 6th Conference on Integrated Design & Process Technology, Pasadena, California, USA. June 23 – 30, 2002 (IDPT 2002)
- Riebisch, M. (2003). Towards a more precise definition of feature models. In: *Modelling Variability for Object-Oriented Product Lines*. BookOnDemand Publ. Co, Norderstedt , pp. 64-76
- Riebisch, M., Streitferdt, M. Pashov, I. (2004). Modeling Variability for Object-Oriented Product Lines. In: Frank,B, Alejandro, B., Mariano, M. (Ed.): *Object-Oriented Technology. ECOOP 2003 Workshop Reader*. Springer, Lecture Notes in Computer Science , vol. 3013, pp. 165 – 178
- Robak, S. (2003). Modeling Variability for Software Product Families. In: Riebisch, M.; Coplien, J.O.; Streitferdt, D. (eds.): *Modelling Variability for Object-Oriented Product Lines*. Publ. Co., Norderstedt, ISBN 3-8330-0779-6. pp. 32-41
- Robillard., P. N. (1999). The role of knowledge in software development. In *Commun. ACM* 42, 1 (January 1999), pp. 87-92. DOI=10.1145/291469.291476
- Rosenmüller, M., Siegmund, N., Schirmeier, H., Sincero, J., Apel, S., Leich, T., Spinczyk, O. , Saake, G. (2008). FAME-DBMS: Tailor-made Data Management Solutions for Embedded Systems. In: *Proceedings of EDBT Workshop on Software Engineering for Tailor-made Data Management (SETMDM)*, pages 1–6. ACM Press.
- Rosenmüller, M., Apel, S., Leich, T., Saake, G. (2009) Tailor-made data management for embedded systems: A case study on Berkeley DB. *Data & Knowledge Engineering* ,Vol 68, Issue 12, pp. 1493-1512.
- Rosenmüller, M., Siegmund, N., Thüm, Th., Saake, G. (2011). Multi-Dimensional Variability Modeling. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pp. 11-20. Namur, Belgium. ACM Press, Jan. 2011.
- Russell, S.J., and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach* (2 ed.). Pearson Education. ISBN:0137903952
- Sabetzadeh, M., Nejati, S., Liaskos, S., Easterbrook, S.M, Chechik, M. (2007). Consistency Checking of Conceptual Models via Model Merging. In: *Proceedings of IEEE International Conference on Requirements Engineering (RE'07)*: pp.221-230
- Savolainen, J., Kuusela, J., Mannion, M., Vehkomäki, T. (2008). Combining Different Product Line Models to Balance Needs of Product Differentiation and Reuse. *ICSR 2008*. pp.116-129

- Schmid, K. and John, I.. (2004). A customizable approach to full lifecycle variability management. *Sci. Comput. Program.* 53, 3 (December 2004), 259-284. DOI=10.1016/j.scico.2003.04.002
- Schmid, K., Rabiser, R., Grünbacher, P. (2011). A comparison of decision modeling approaches in product lines. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems (VaMoS '11)*. ACM, New York, NY, USA, pp. 119-126. DOI=10.1145/1944892.1944907
- Schobbens, P., Heymans, P., Trigaux, J-C., Bontemps, Y. (2007). Generic semantics of feature diagrams, In: *Computer Networks*, Volume 51, Issue 2, pp. 456-479
- Schreiber, G. (2008). Knowledge Engineering, *Handbook of Knowledge Representation*, chapter 25, Eds F. van Harmelen, V. Lifschitz and B. Porter, Elsevier
- Schreiber, G., Akkermans, H., Anjewierden, A., de Hoog, R., Shadbolt, N., Van de Velde, W., Wielinga, B. (2000). *Knowledge Engineering and Management: The CommonKADS Methodology*. MIT Press, ISBN 0262193000..
- Schroeter, J., Lochau, M., Winkelmann, T. (2012). Multi-perspectives on Feature Models. In *Proceedings of the 15th international conference on Model Driven Engineering Languages and Systems (MODELS'12)*, Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson (Eds.). Springer-Verlag, Berlin, Heidelberg, pp. 252-268. DOI=10.1007/978-3-642-33666-9_17
- Sebastian, A., Noy, N. F., Tudorache, T., Musen, M. A. (2008). A Generic Ontology For Collaborative Ontology-Development Workflows, 16th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2008), Catania, Italy, Springer. Published in 2008
- Segura, S., Benavides, D., Trinidad, P., Ruiz-Cortés, A. (2007) Automated Merging of Feature Models using Graph Transformations. In: *Post-proceedings Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'07)*
- Sellier, D. and Mannion, M. (2007). Visualising Product Line Requirement Selection Decision Inter-dependencies. In *Proceedings of the Second International Workshop on Requirements Engineering Visualization (REV '07)*. IEEE Computer Society, Washington, DC, USA
- Shapiro, S. (2001). *Classical Logic II: Higher Order Logic*. In Lou Goble, ed., *The Blackwell Guide to Philosophical Logic*. Blackwell, ISBN 0-631-20693-0
- Siegmund, N., Kästner, C., Rosenmüller, M., Heidenreich, F., Apel, S., Saake, G. (2009). Bridging the Gap between Variability in Client Application and Database Schema. *BTW 2009*, pp. 297-306.
- Simon, H.A. (1981). *The Sciences of the Artificial* (2nd Ed.). MIT Press, Cambridge, MA, USA. ISBN:0-262-69191-4
- Sinnema, M., and Deelstra, S. (2007). Classifying Variability Modeling Techniques, In: *Elsevier Journal on Information and Software Technology*, vol. 49, Issue 7, pp. 717 -739
- Sirin, E., Parsia, B., CuencaGrau, B., Kalyanpur, A., Katz, Y. (2007). Pellet: a practical OWL-DL reasoner. *J. Web Semant.* Vol. 5(2), pp.51–53
- Software Productivity Consortium Services Corporation, Technical Report SPC-92019-CMC. Reuse-Driven Software Processes Guidebook, Version 02.00.03, 1993. <http://www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&doc=GetTRDoc.pdf&AD=ADA273644>
- Sowa, J. F. (1992). Semantic networks, In: *Encyclopedia of Artificial Intelligence*, edited by S. C. Shapiro, Wiley, New York, 1987; revised and extended for the second edition, 1992
- Space ontology, <http://sweet.jpl.nasa.gov/ontology/space.owl>, last visit 1/3/2011
- Spyns, P. (2005). Object role modelling for ontology engineering in the DOGMA framework. In *Proceedings of the 2005 OTM Confederated international conference on On the Move to Meaningful Internet Systems (OTM'05)*, Robert Meersman, Zahir Tari, and Pilar Herrero (Eds.). Springer-Verlag, Berlin, Heidelberg, 710-719. DOI=10.1007/11575863_90 http://dx.doi.org/10.1007/11575863_90

List of References

- Srivastava, B., and Koehler, J. (2003). Web service composition - current solutions and open problems, In: Proceedings of ICAPS 2003
- Svahnberg, M., and Bosch, J. (1999). Evolution in software product lines: Two cases. In: Journal of Software Maintenance: Research and Practice vol. 11 , Issue 6
- Svahnberg, M., van Gorp, J., Bosch, J. (2005). A Taxonomy of Variability Realization Techniques. in Software Practice & Experience, 35(8). pp.705-754.
- Synthesis. (1993). Software Productivity Consortium Services Corporation, "Reuse- Driven Software Processes," Technical Report SPC-92019-CMC, Version 02.00.03, November 1993
- Tarr, P., Ossher, H., Harrison, W., Sutton, J. S. M.(1999). N Degrees of Separation: Multi-Dimensional Separation of Concerns. In Proc. Int'l. Conf. Software Engineering (ICSE), pp. 107–119. IEEE CS
- Tesanovic, A., Sheng, K., Hansson, J. (2004) Application-Tailored Database Systems: a Case of Aspects in an Embedded Database. In: Proceedings of the 8th International Database Engineering and Applications Symposium (IDEAS'04), IEEE Computer Society.
- Time Ontology in OWL, <http://www.w3.org/TR/owl-time/>, 2006, last visit 1/3/ 2011
- Tracz , W., Coglianese, L., and Young, P. (1993). Domain specific SW architecture engineering. Software Engineering Notes, vol.18(2)
- Trinidad, P., Cortés, A. R., Benavides, D., Segura, S. (2008).Three-Dimensional Feature Diagrams Visualization. In Proceedings of Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, pp. 295-302.
- Tsarkov, D., Horrocks, I. (2006). FaCT++ description logic reasoner: system description. In: Proceedings of the Third International Joint Conference on Automated Reasoning (IJCAR 2006), Seattle. Lecture Notes in Artificial Intelligence, vol. 4130, pp. 292–297. Springer, Berlin
- Tun, T. T., Boucher, Q., Classen, A., Hubaux, A., Heymans, P. (2009) Relating requirements and feature configurations: a systematic approach. In Proceedings of the 13th International Software Product Line Conference (SPLC '09). Carnegie Mellon University, Pittsburgh, PA, USA, 201-210
- Van Gorp, J., Bosch, J., and Svahnberg, M. (2001). On the Notion of Variability in Software Product Lines. In Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA '01). IEEE Computer Society, Washington, DC, USA
- van Lamsweerde, A. (2009) Requirements Engineering From System Goals to UML Models to Software Specifications, ISBN 978-0-470-01270-3, January 2009
- van Ommering ,R. (2000). Beyond product families: building a product population. In Proceedings of the International Workshop on Software Architectures for Product Families (IW-SAPF-3), Springer, Las Palmas de Gran Canaria, Spain, 2000, pp. 187–198.
- van Ommering, R. (2002). Building Product Populations with Software Components. In Proceedings of 24th International Conference on Software Engineering, Orlando, Florida, September 2002, pp. 255-265.
- van Ommering, R. C., Bosch, J. (2002). Widening the Scope of Software Product Lines - From Variation to Composition. In Proceedings of the Second International Conference on Software Product Lines (SPLC 2), Gary J. Chastek (Ed.). Springer-Verlag, London, UK, UK, pp.328-347.
- van Ommering, R. (2005). Software Reuse in Product Populations. IEEE Trans. Softw. Eng. 31, 7 (July 2005), pp. 537-550. DOI=10.1109/TSE.2005.84
- von der Massen, T., Lichter, H. (2004). Deficiencies in feature models. In: T. Mannisto, J. Bosch (Eds.), Workshop on Software Variability Management for Product Derivation - Towards Tool Support
- W3C, The Semantic Web Made Easy, <http://www.w3.org/RDF/Metalog/docs/sw-easy>, last visit1/3/ 2011

List of References

- Walter, T., Parreiras, F.S., Gröner, G., Wende, C. (2010). OWLizing: Transforming Software Models to Ontologies. In Proceedings of Ontology-Driven Software Engineering, ODISE'10, ACM
- Wand, Y. and Weber, R. (1993). On the Ontological Expressiveness of Information Systems Analysis and Design Grammars, *Journal of Information Systems*, vol. 3, pp. 217-237.
- Wand, Y. and Weber, R. (1993). On the ontological expressiveness of information systems analysis and design grammars. *Journal of Information Systems*, 3, pp.217-237.
- Wand, Y., and Weber, R. (2002). Research Commentary: Information Systems and Conceptual Modeling — A Research Agenda. *Info. Sys. Research* 13, 4 (December 2002), pp.363-376. DOI=10.1287/isre.13.4.363.69.
- Wang, J. A. (2000). Towards component-based software engineering. In Proceedings of the eighth annual consortium on Computing in Small Colleges Rocky Mountain conference. Consortium for Computing Sciences in Colleges, USA, pp. 177-189.
- Wang, H., Li, Y., Sun, J., Zhang, H., Pan, J. (2005). A semantic web approach to feature modeling and verification. In: Proceedings of Workshop on Semantic Web Enabled Software Engineering (SWESE'05)
- Weiler, T. (2003). Modelling Architectural Variability for Software Product Lines. In Proceedings of Software Variability Management ICSE2003 Workshop, SVM 2003, Groningen, IWI 2003-7-01, The Netherlands, pp.5-12.
- Welzer, T., Stiglic, B., Rozman, I., Družovec, M. (1999) Reusable Conceptual Models as a Support for the Higher Information Quality, ICPQR99
- Wielinga, B. J., Van de Velde, W., Schreiber, A. Th., Akkermans, J. M. (1992). The KADS knowledge modelling approach. In R. Mizoguchi, H. Motoda, J. Boose, B. Gaines, and R. Quinlan, editors, Proceedings of the 2nd Japanese Knowledge Acquisition for Knowledge-Based Systems Workshop, pages 23-42. Hitachi, Advanced Research Laboratory, Hatoyama, Saitama, Japan
- Witte, R., Zhang, Y., Rilling, J. (2007). Empowering Software Maintainers with Semantic Web Technologies. In Proceedings of the 4th European Semantic Web Conference, pp. 37-52. Springer.
- Woods, E. (2004). Experiences Using Viewpoints for Information Systems Architecture: An Industrial Experience Report. In: Proceedings of EWSA 2004: pp. 182-193, 2004
- Ye, H.; Liu, H. (2005). Approach to modelling feature variability and dependencies in software product lines. In: *Software, IEE Proceedings - Volume 152, Issue 3*, pp.101-109
- Zave, P. (2004). FAQ Sheet on Feature Interaction, <http://www.research.att.com/~pamela/faq.html>, last visit 1/2/2010
- Ziadi, T., Héluët, L., Jézéquel, J.-M. (2003). Towards a UML Profile for Software Product Lines, In Software Product-Family Engineering, 5th International Workshop. Seana / Italy, 2003, Springer LNCS 3014, pp. 129-139, Springer-Verlag.

composed feature or an elementary feature. This is represented in the conceptual model by means of the Object Type *Feature* having two subtypes *Elementary Feature* and *Composed Feature*. A composed feature is a feature that is further decomposed into other finer grained features. An elementary feature is a leaf feature, which is not further decomposed. Based on how features contribute to the software functionalities, features are classified according to their role in the system (several classifications exist). We merged the classification of features and choose the most common classifications ones: functional feature, parameter feature, external feature, and interface feature. A *functional feature* is a feature that contributes to the programs functionality. A *parameter feature* is a feature that represents some parameterized characteristic in the software and is usually associated with a value. An *external feature* is a feature that represents an external attribute which interacts with the software or contributes to it. An interface feature is a feature on the boundary of the software system and the external surroundings; furthermore it provides a connection point between the software and its surroundings (surroundings could be users or other devices). The different features are represented in the conceptual model by means of the subtypes *Functional Feature*, *Parameter Feature*, *External Feature*, and *Interface Feature*.

A feature can have a value associated with it; this is represented by the Object Type Feature Value. For example, a feature that calculates the shipping cost in E-shop software is shown in figure A.2. The shipping cost consists of an optional part fixed shipping tax and a mandatory part shipping fees; fixed shipping tax is a feature that is associated with a feature value, which has the value 5. A feature can have one or more feature attributes. Feature attributes are represented in our conceptual model by means of the object type Attribute. There are two types of attributes: Value Attributes and Reference Attributes. A Value Attribute is associated with a value; therefore it has an Attribute value attached to it. For example, in figure 3 rate, source and destination represent three value attributes, as each of them is associated with a value. Reference Attributes are attributes that refer to another feature. For example, in the E-shop software, we could assume there is a purchase order feature; this feature would have an attribute shipping cost that refers to the shipping cost feature (given in figure A.2). In addition, a feature can have a Feature Value constraint; in this case instead of having a specific value, the feature value is determined by a constraint. The object type Value Constraint represents this type of constraint. It must be noted that a feature can either have a value constraint or a value and not both; this is indicated by the exclusion relation between the two roles.

A feature can be part of a composition (AND, OR, Alternative) or be standalone (Mandatory, Optional). Therefore we have defined the object type *Feature Composition* that is composed of a number of *Feature in Composition*. *Feature Composition* has three subtypes *AND Composition*, *Alternative Composition*, and *OR Composition*, which correspond to the relations *And*, *Alternative*, and *Or* respectively. *Feature Composition* represents the feature group forming the composition; it is a mapping of the relation branches in feature models.

Features that are in a composition (represented by the *Feature in Composition* object type) have (in addition to their composition) a type that indicates the type of the feature irrelevant to how it is composed; e.g., can be mandatory or optional. This is represented by the subtypes *Mandatory Feature*, *Optional Feature*, *OR Feature*, and *Alternative Feature*. In order to ensure consistency, features that are members of an *AND Composition* should be given the type *mandatory* (i.e. every member of the group has an *and* relation with all other members).

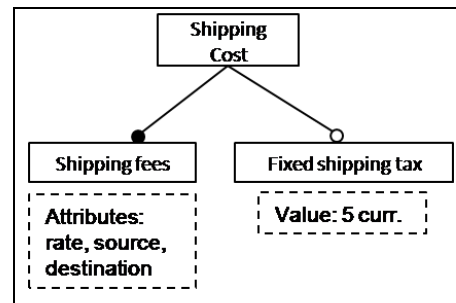


Figure. A.3. Feature model showing shipping cost example

The number of times a feature is allowed to occur in the software product is expressed by the cardinality relation. There are two types of cardinalities allowed, *Clone Cardinality* and *Composition Cardinality*. The *Clone Cardinality* represents the number of copies (instances) of a feature that could coexist in the software product; it is only valid for mandatory features. The *Composition Cardinality* refers to the number of allowed sub-features of a specific composition in the software, which makes it only valid for features belonging to an *Or Composition*. Features of an *Alternative Composition* will always have a cardinality of one, while features of an *AND Composition* will always have a cardinality that is equal to the number of features in the composition. *Cardinality* has a maximum upper bound represented by the object type *Upper_Bound* and a minimum lower bound represented by the object type *Lower_Bound* for expressing the existence/coexistence of features in the software. Furthermore, alternative to defining a range for the number of features that could exist in a product an exact number can be given; this is represented by the object type *Exact*. Having an exact cardinality excludes both having an upper bound and a lower bound cardinality, this is indicated by the exclusive relation between the roles: *Exact_cardinality*, *With_lower_bound* and *Exact_cardinality*, *With_upper_bound*.

Appendix B

FAM Ontology in OWL Functional Syntax

```

Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
Prefix(swrlb:=<http://www.w3.org/2003/11/swrlb#>)
Prefix(owl2xml:=<http://www.w3.org/2006/12/owl2-xml#>)
Prefix(owl:=<http://www.w3.org/2002/07/owl#>)
Prefix(:=<http://wise.vub.ac.be/Members/lamia/variability/Feature_Assembly/FAM.owl#>)
Prefix(xml:=<http://www.w3.org/XML/1998/namespace>)
Prefix(rdf:=<http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
Prefix(swrl:=<http://www.w3.org/2003/11/swrl#>)
Prefix(rdfs:=<http://www.w3.org/2000/01/rdf-schema#>)

Ontology(<http://wise.vub.ac.be/Members/lamia/variability/Feature_Assembly/FAM.owl>

Declaration(Class(:Abstract_Feature))
EquivalentClasses(:Abstract_Feature ObjectIntersectionOf(:Abstract_Feature
ObjectAllValuesFrom(:has_Option :Option_Feature)))
SubClassOf(:Abstract_Feature :Feature)
DisjointClasses(:Abstract_Feature :Concrete_Feature)
Declaration(Class(:Binding_Time))
DisjointClasses(:Binding_Time :FCardinality)
DisjointClasses(:Binding_Time :Feature)
DisjointClasses(:Binding_Time :Perspective)
DisjointClasses(:Binding_Time :Priority)
DisjointClasses(:Binding_Time :Stakeholder)
DisjointClasses(:Binding_Time :Variability)
Declaration(Class(:Business_Analyst))
SubClassOf(:Business_Analyst :Stakeholder)
Declaration(Class(:Cardinality_Error))
SubClassOf(:Cardinality_Error :Error)
Declaration(Class(:Client))
SubClassOf(:Client :Stakeholder)
Declaration(Class(:Concrete_Feature))
EquivalentClasses(:Concrete_Feature ObjectIntersectionOf(:Concrete_Feature
ObjectAllValuesFrom(:mandatory_Composition ObjectUnionOf(:Concrete_Feature
:Abstract_Feature))))
EquivalentClasses(:Concrete_Feature ObjectIntersectionOf(:Concrete_Feature
ObjectAllValuesFrom(:optional_Composition ObjectUnionOf(:Concrete_Feature
:Abstract_Feature))))
SubClassOf(:Concrete_Feature :Feature)
DisjointClasses(:Concrete_Feature :Abstract_Feature)
Declaration(Class(:Cyclic_Error))
SubClassOf(:Cyclic_Error :Error)
Declaration(Class(:Developer))
SubClassOf(:Developer :Stakeholder)
Declaration(Class(:Domain_Expert))
SubClassOf(:Domain_Expert :Stakeholder)
Declaration(Class(:Error))
SubClassOf(:Error owl:Thing)
DisjointClasses(:Error :Priority)
Declaration(Class(:FCardinality))
EquivalentClasses(:FCardinality ObjectIntersectionOf(:FCardinality DataMaxCardinality(1
:max xsd:int)))
EquivalentClasses(:FCardinality ObjectIntersectionOf(:FCardinality DataMaxCardinality(1
:min xsd:int)))
DisjointClasses(:FCardinality :Binding_Time)
DisjointClasses(:FCardinality :Feature)
DisjointClasses(:FCardinality :Perspective)
DisjointClasses(:FCardinality :Priority)

```

```

DisjointClasses(:FCardinality :Stakeholder)
Declaration(Class(:Feature))
EquivalentClasses(:Feature                               ObjectIntersectionOf(:Feature
ObjectAllValuesFrom(:has_Stakeholder :Stakeholder)))
EquivalentClasses(:Feature                               ObjectIntersectionOf(:Feature       ObjectMaxCardinality(1
:has_Owner :Stakeholder)))
DisjointClasses(:Feature :Binding_Time)
DisjointClasses(:Feature :FCardinality)
DisjointClasses(:Feature :Perspective)
DisjointClasses(:Feature :Priority)
DisjointClasses(:Feature :Stakeholder)
Declaration(Class(:Functional))
SubClassOf(:Functional :Perspective)
Declaration(Class(:Graphical_User_Interface))
SubClassOf(:Graphical_User_Interface :Perspective)
Declaration(Class(:Hardware_Interface))
SubClassOf(:Hardware_Interface :Perspective)
Declaration(Class(:Inconsistency))
SubClassOf(:Inconsistency :Error)
Declaration(Class(:Localization))
SubClassOf(:Localization :Perspective)
Declaration(Class(:Marketing))
SubClassOf(:Marketing :Stakeholder)
Declaration(Class(:Modeller))
SubClassOf(:Modeller :Stakeholder)
Declaration(Class(:Non_Functional))
SubClassOf(:Non_Functional :Perspective)
Declaration(Class(:Option_Feature))
EquivalentClasses(:Option_Feature :Variant)
EquivalentClasses(:Option_Feature                               ObjectIntersectionOf(:Option_Feature
ObjectAllValuesFrom(:option_Of :Abstract_Feature)))
SubClassOf(:Option_Feature :Feature)
Declaration(Class(:Persistent))
SubClassOf(:Persistent :Perspective)
Declaration(Class(:Perspective))
DisjointClasses(:Perspective :Binding_Time)
DisjointClasses(:Perspective :FCardinality)
DisjointClasses(:Perspective :Feature)
DisjointClasses(:Perspective :Priority)
DisjointClasses(:Perspective :Stakeholder)
Declaration(Class(:Priority))
DisjointClasses(:Priority :Binding_Time)
DisjointClasses(:Priority :Error)
DisjointClasses(:Priority :FCardinality)
DisjointClasses(:Priority :Feature)
DisjointClasses(:Priority :Perspective)
DisjointClasses(:Priority :Stakeholder)
DisjointClasses(:Priority :Variability)
Declaration(Class(:Project_Manager))
SubClassOf(:Project_Manager :Stakeholder)
Declaration(Class(:Redundancy))
SubClassOf(:Redundancy :Error)
Declaration(Class(:Sales))
SubClassOf(:Sales :Stakeholder)
Declaration(Class(:Stakeholder))
DisjointClasses(:Stakeholder :Binding_Time)
DisjointClasses(:Stakeholder :FCardinality)
DisjointClasses(:Stakeholder :Feature)
DisjointClasses(:Stakeholder :Perspective)
DisjointClasses(:Stakeholder :Priority)
Declaration(Class(:System))
SubClassOf(:System :Perspective)
Declaration(Class(:Task))
SubClassOf(:Task :Perspective)
Declaration(Class(:Testers))
SubClassOf(:Testers :Stakeholder)
Declaration(Class(:User))
SubClassOf(:User :Perspective)
Declaration(Class(:Variability))
DisjointClasses(:Variability :Binding_Time)
DisjointClasses(:Variability :Priority)
Declaration(Class(:Variant))
EquivalentClasses(:Variant :Option_Feature)

```



```

SubClassOf(:Variant :Variability)
Declaration(Class(:Variation_Point))
SubClassOf(:Variation_Point :Variability)
Declaration(Class(owl:Thing))
Declaration(ObjectProperty(:FTFC))
ObjectPropertyDomain(:FTFC :Feature)
ObjectPropertyRange(:FTFC :Feature)
Declaration(ObjectProperty(:belongs_To))
ObjectPropertyDomain(:belongs_To :Feature)
ObjectPropertyRange(:belongs_To :Perspective)
Declaration(ObjectProperty(:composition))
ObjectPropertyDomain(:composition :Concrete_Feature)
ObjectPropertyRange(:composition ObjectUnionOf(:Abstract_Feature :Concrete_Feature
ObjectComplementOf(:Option_Feature)))
Declaration(ObjectProperty(:cyclic))
SymmetricObjectProperty(:cyclic)
ObjectPropertyDomain(:cyclic :Cyclic_Error)
ObjectPropertyRange(:cyclic :Cyclic_Error)
Declaration(ObjectProperty(:excludes))
SubObjectPropertyOf(:excludes :FTFC)
SymmetricObjectProperty(:excludes)
Declaration(ObjectProperty(:has_Binding_Time))
FunctionalObjectProperty(:has_Binding_Time)
ObjectPropertyDomain(:has_Binding_Time :Feature)
ObjectPropertyRange(:has_Binding_Time :Binding_Time)
Declaration(ObjectProperty(:has_Cardinality))
FunctionalObjectProperty(:has_Cardinality)
ObjectPropertyDomain(:has_Cardinality :Abstract_Feature)
ObjectPropertyRange(:has_Cardinality :FCardinality)
Declaration(ObjectProperty(:has_Option))
InverseObjectProperties(:has_Option :option_Of)
ObjectPropertyDomain(:has_Option :Abstract_Feature)
ObjectPropertyRange(:has_Option :Option_Feature)
Declaration(ObjectProperty(:has_Owner))
ObjectPropertyDomain(:has_Owner :Feature)
ObjectPropertyRange(:has_Owner :Stakeholder)
Declaration(ObjectProperty(:has_Priority))
FunctionalObjectProperty(:has_Priority)
Declaration(ObjectProperty(:has_Stakeholder))
ObjectPropertyDomain(:has_Stakeholder :Feature)
ObjectPropertyRange(:has_Stakeholder :Stakeholder)
Declaration(ObjectProperty(:inconsistent))
SymmetricObjectProperty(:inconsistent)
ObjectPropertyDomain(:inconsistent :Inconsistency)
ObjectPropertyRange(:inconsistent :Inconsistency)
Declaration(ObjectProperty(:mandatory_Composition))
SubObjectPropertyOf(:mandatory_Composition :composition)
ObjectPropertyDomain(:mandatory_Composition :Concrete_Feature)
ObjectPropertyRange(:mandatory_Composition ObjectUnionOf(:Abstract_Feature
:Concrete_Feature ObjectComplementOf(:Option_Feature)))
Declaration(ObjectProperty(:option_Of))
InverseObjectProperties(:has_Option :option_Of)
ObjectPropertyDomain(:option_Of :Option_Feature)
ObjectPropertyRange(:option_Of :Abstract_Feature)
Declaration(ObjectProperty(:optional_Composition))
SubObjectPropertyOf(:optional_Composition :composition)
ObjectPropertyDomain(:optional_Composition :Concrete_Feature)
ObjectPropertyRange(:optional_Composition ObjectUnionOf(:Abstract_Feature
:Concrete_Feature ObjectComplementOf(:Option_Feature)))
Declaration(ObjectProperty(:redundant))
SymmetricObjectProperty(:redundant)
ObjectPropertyDomain(:redundant :Redundancy)
ObjectPropertyRange(:redundant :Redundancy)
Declaration(ObjectProperty(:requires))
SubObjectPropertyOf(:requires :FTFC)
TransitiveObjectProperty(:requires)
Declaration(ObjectProperty(:same))
SubObjectPropertyOf(:same :FTFC)
SymmetricObjectProperty(:same)
Declaration(ObjectProperty(:uses))
SubObjectPropertyOf(:uses :FTFC)
TransitiveObjectProperty(:uses)
Declaration(DataProperty(:has_Description))

```

```

DataPropertyDomain(:has_Description :Feature)
DataPropertyRange(:has_Description xsd:string)
Declaration(DataProperty(:max))
DataPropertyDomain(:max :FCardinality)
Declaration(DataProperty(:min))
DataPropertyDomain(:min :FCardinality)
Declaration(NamedIndividual(:Analysis))
ClassAssertion(:Binding_Time :Analysis)
ClassAssertion(owl:Thing :Analysis)
Declaration(NamedIndividual(:Compilation))
ClassAssertion(:Binding_Time :Compilation)
ClassAssertion(owl:Thing :Compilation)
Declaration(NamedIndividual(:Design))
ClassAssertion(:Binding_Time :Design)
ClassAssertion(owl:Thing :Design)
Declaration(NamedIndividual(:High))
ClassAssertion(:Priority :High)
ClassAssertion(owl:Thing :High)
Declaration(NamedIndividual(:Implementation))
ClassAssertion(:Binding_Time :Implementation)
ClassAssertion(owl:Thing :Implementation)
Declaration(NamedIndividual(:Installation))
ClassAssertion(:Binding_Time :Installation)
ClassAssertion(owl:Thing :Installation)
Declaration(NamedIndividual(:Low))
ClassAssertion(:Priority :Low)
ClassAssertion(owl:Thing :Low)
Declaration(NamedIndividual(:Medium))
ClassAssertion(:Priority :Medium)
ClassAssertion(owl:Thing :Medium)
Declaration(NamedIndividual(:None))
ClassAssertion(:Priority :None)
ClassAssertion(owl:Thing :None)
Declaration(NamedIndividual(:Startup))
ClassAssertion(:Binding_Time :Startup)
ClassAssertion(owl:Thing :Startup)
Declaration(NamedIndividual(:Top))
ClassAssertion(:Priority :Top)
ClassAssertion(owl:Thing :Top)
Declaration(AnnotationProperty(:Dependency_Reason))
Declaration(AnnotationProperty(:Dependency_Owner))
DLSafeRule(Body(ObjectPropertyAtom(:has_Option Variable(<urn:swrl#x>
Variable(<urn:swrl#y>)) ClassAtom(:Abstract_Feature
Variable(<urn:swrl#x>))) Head(ClassAtom(:Variation_Point Variable(<urn:swrl#x>)))
DLSafeRule(Body(ClassAtom(:Abstract_Feature Variable(<urn:swrl#x>))
ObjectPropertyAtom(:excludes Variable(<urn:swrl#x>)) Variable(<urn:swrl#y>))
ObjectPropertyAtom(:has_Option Variable(<urn:swrl#x>))
Variable(<urn:swrl#z>)) Head(ObjectPropertyAtom(:excludes Variable(<urn:swrl#z>
Variable(<urn:swrl#y>)))
DLSafeRule(Body(ObjectPropertyAtom(:uses Variable(<urn:swrl#x>)) Variable(<urn:swrl#y>))
ObjectPropertyAtom(:excludes Variable(<urn:swrl#x>))
Variable(<urn:swrl#y>)) Head(ObjectPropertyAtom(:inconsistent Variable(<urn:swrl#x>
Variable(<urn:swrl#y>)))
DLSafeRule(Body(ClassAtom(:Concrete_Feature Variable(<urn:swrl#x>))
ObjectPropertyAtom(:excludes Variable(<urn:swrl#x>)) Variable(<urn:swrl#y>))
ObjectPropertyAtom(:mandatory_Composition Variable(<urn:swrl#x>))
Variable(<urn:swrl#z>)) Head(ObjectPropertyAtom(:excludes Variable(<urn:swrl#z>
Variable(<urn:swrl#y>)))
DLSafeRule(Body(ObjectPropertyAtom(:uses Variable(<urn:swrl#y>)) Variable(<urn:swrl#x>))
ObjectPropertyAtom(:uses Variable(<urn:swrl#x>))
Variable(<urn:swrl#y>)) Head(ObjectPropertyAtom(:cyclic Variable(<urn:swrl#x>
Variable(<urn:swrl#y>)))
DLSafeRule(Body(DataPropertyAtom(:max Variable(<urn:swrl#x>)) Variable(<urn:swrl#y>))
DataPropertyAtom(:min Variable(<urn:swrl#x>)) Variable(<urn:swrl#z>))
BuiltInAtom(swrlb:greaterThan Variable(<urn:swrl#x>)) Variable(<urn:swrl#z>))
Variable(<urn:swrl#y>)) Head(ClassAtom(:Cardinality_Error Variable(<urn:swrl#x>)))
DLSafeRule(Body(ObjectPropertyAtom(:requires Variable(<urn:swrl#x>))
Variable(<urn:swrl#y>)) ObjectPropertyAtom(:excludes Variable(<urn:swrl#x>))
Variable(<urn:swrl#y>)) Head(ObjectPropertyAtom(:inconsistent Variable(<urn:swrl#x>
Variable(<urn:swrl#y>)))
DLSafeRule(Body(ObjectPropertyAtom(:uses Variable(<urn:swrl#x>)) Variable(<urn:swrl#y>))
ObjectPropertyAtom(:requires Variable(<urn:swrl#x>))

```

```

Variable(<urn:swrl#y>))Head(ObjectPropertyAtom(:redundant      Variable(<urn:swrl#x>)
Variable(<urn:swrl#y>)))
DLSafeRule(Body(ObjectPropertyAtom(:optional_Composition      Variable(<urn:swrl#x>)
Variable(<urn:swrl#y>))          ClassAtom(:Concrete_Feature
Variable(<urn:swrl#x>))Head(ClassAtom(:Variation_Point Variable(<urn:swrl#x>)))
DLSafeRule(Body(ClassAtom(:Concrete_Feature      Variable(<urn:swrl#x>))
ObjectPropertyAtom(:excludes      Variable(<urn:swrl#x>)      Variable(<urn:swrl#y>))
ObjectPropertyAtom(:optional_Composition      Variable(<urn:swrl#x>)
Variable(<urn:swrl#z>))Head(ObjectPropertyAtom(:excludes      Variable(<urn:swrl#z>)
Variable(<urn:swrl#y>)))
)

```


Appendix C

OWL DL Description Logic Representation

Abstract Syntax	DL Syntax	Semantics
Class(<i>A</i> partial $C_1 \dots C_n$)	$A \sqsubseteq C_1 \sqcap \dots \sqcap C_n$	$A^{\mathcal{I}} \subseteq C_1^{\mathcal{I}} \cap \dots \cap C_n^{\mathcal{I}}$
Class(<i>A</i> complete $C_1 \dots C_n$)	$A = C_1 \sqcap \dots \sqcap C_n$	$A^{\mathcal{I}} = C_1^{\mathcal{I}} \cap \dots \cap C_n^{\mathcal{I}}$
EnumeratedClass(<i>A</i> $o_1 \dots o_n$)	$A = \{o_1, \dots, o_n\}$	$A^{\mathcal{I}} = \{o_1^{\mathcal{I}}, \dots, o_n^{\mathcal{I}}\}$
SubClassOf($C_1 C_2$)	$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$
EquivalentClasses($C_1 \dots C_n$)	$C_1 = \dots = C_n$	$C_1^{\mathcal{I}} = \dots = C_n^{\mathcal{I}}$
DisjointClasses($C_1 \dots C_n$)	$C_i \sqcap C_j = \perp, i \neq j$	$C_i^{\mathcal{I}} \cap C_j^{\mathcal{I}} = \emptyset, i \neq j$
Datatype(<i>D</i>)		$D^{\mathcal{I}} \subseteq \Delta_D^{\mathcal{I}}$
DatatypeProperty(<i>U</i> super($U_1 \dots U_n$) domain($C_1 \dots C_m$) range($D_1 \dots D_l$) [Functional])	$U \sqsubseteq U_i$ $\geq 1 U \sqsubseteq C_i$ $\top \sqsubseteq \forall U.D_i$ $\top \sqsubseteq \leq 1 U$	$U^{\mathcal{I}} \subseteq U_i^{\mathcal{I}}$ $U^{\mathcal{I}} \subseteq C_i^{\mathcal{I}} \times \Delta_D^{\mathcal{I}}$ $U^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times D_i^{\mathcal{I}}$ $U^{\mathcal{I}}$ is functional
SubPropertyOf($U_1 U_2$)	$U_1 \sqsubseteq U_2$	$U_1^{\mathcal{I}} \subseteq U_2^{\mathcal{I}}$
EquivalentProperties($U_1 \dots U_n$)	$U_1 = \dots = U_n$	$U_1^{\mathcal{I}} = \dots = U_n^{\mathcal{I}}$
ObjectProperty(<i>R</i> super($R_1 \dots R_n$) domain($C_1 \dots C_m$) range($C_1 \dots C_l$) [inverseOf(R_0) [Symmetric] [Functional] [InverseFunctional] [Transitive]])	$R \sqsubseteq R_i$ $\geq 1 R \sqsubseteq C_i$ $\top \sqsubseteq \forall R.C_i$ $R = (\neg R_0)$ $R = (\neg R)$ $\top \sqsubseteq \leq 1 R$ $\top \sqsubseteq \leq 1 R^-$ $Tr(R)$	$R^{\mathcal{I}} \subseteq R_i^{\mathcal{I}}$ $R^{\mathcal{I}} \subseteq C_i^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times C_i^{\mathcal{I}}$ $R^{\mathcal{I}} = (R_0^{\mathcal{I}})^-$ $R^{\mathcal{I}} = (R^{\mathcal{I}})^-$ $R^{\mathcal{I}}$ is functional $(R^{\mathcal{I}})^-$ is functional $R^{\mathcal{I}} = (R^{\mathcal{I}})^+$
SubPropertyOf($R_1 R_2$)	$R_1 \sqsubseteq R_2$	$R_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$
EquivalentProperties($R_1 \dots R_n$)	$R_1 = \dots = R_n$	$R_1^{\mathcal{I}} = \dots = R_n^{\mathcal{I}}$
AnnotationProperty(<i>S</i>)		
Individual(<i>o</i> type($C_1 \dots C_n$) value($R_1 o_1 \dots R_n o_n$) value($U_1 v_1 \dots U_n v_n$))	$o \in C_i$ $\langle o, o_i \rangle \in R_i$ $\langle o, v_i \rangle \in U_i$	$o^{\mathcal{I}} \in C_i^{\mathcal{I}}$ $\langle o^{\mathcal{I}}, o_i^{\mathcal{I}} \rangle \in R_i^{\mathcal{I}}$ $\langle o^{\mathcal{I}}, v_i^{\mathcal{I}} \rangle \in U_i^{\mathcal{I}}$
SameIndividual($o_1 \dots o_n$)	$o_1 = \dots = o_n$	$o_i^{\mathcal{I}} = o_j^{\mathcal{I}}$
DifferentIndividuals($o_1 \dots o_n$)	$o_i \neq o_j, i \neq j$	$o_i^{\mathcal{I}} \neq o_j^{\mathcal{I}}, i \neq j$

Figure. B.1. OWL DL Axioms and Facts [Horrocks et. al., 2003]

Abstract Syntax	DL Syntax	Semantics
Descriptions (C)		
A (URI reference)	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
<code>owl:Thing</code>	\top	$\text{owl:Thing}^{\mathcal{I}} = \Delta^{\mathcal{I}}$
<code>owl:Nothing</code>	\perp	$\text{owl:Nothing}^{\mathcal{I}} = \{\}$
<code>intersectionOf($C_1 C_2 \dots$)</code>	$C_1 \sqcap C_2$	$(C_1 \sqcap C_2)^{\mathcal{I}} = C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$
<code>unionOf($C_1 C_2 \dots$)</code>	$C_1 \sqcup C_2$	$(C_1 \sqcup C_2)^{\mathcal{I}} = C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$
<code>complementOf(C)</code>	$\neg C$	$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
<code>oneOf($o_1 \dots$)</code>	$\{o_1, \dots\}$	$\{o_1, \dots\}^{\mathcal{I}} = \{o_1^{\mathcal{I}}, \dots\}$
<code>restriction(R someValuesFrom(C))</code>	$\exists R.C$	$(\exists R.C)^{\mathcal{I}} = \{x \mid \exists y. \langle x, y \rangle \in R^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\}$
<code>restriction(R allValuesFrom(C))</code>	$\forall R.C$	$(\forall R.C)^{\mathcal{I}} = \{x \mid \forall y. \langle x, y \rangle \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
<code>restriction(R hasValue(o))</code>	$R : o$	$(\forall R.o)^{\mathcal{I}} = \{x \mid \langle x, o^{\mathcal{I}} \rangle \in R^{\mathcal{I}}\}$
<code>restriction(R minCardinality(n))</code>	$\geq n R$	$(\geq n R)^{\mathcal{I}} = \{x \mid \#\{\langle y, x \rangle \in R^{\mathcal{I}}\} \geq n\}$
<code>restriction(R maxCardinality(n))</code>	$\leq n R$	$(\leq n R)^{\mathcal{I}} = \{x \mid \#\{\langle y, x \rangle \in R^{\mathcal{I}}\} \leq n\}$
<code>restriction(U someValuesFrom(D))</code>	$\exists U.D$	$(\exists U.D)^{\mathcal{I}} = \{x \mid \exists y. \langle x, y \rangle \in U^{\mathcal{I}} \text{ and } y \in D^{\mathbf{D}}\}$
<code>restriction(U allValuesFrom(D))</code>	$\forall U.D$	$(\forall U.D)^{\mathcal{I}} = \{x \mid \forall y. \langle x, y \rangle \in U^{\mathcal{I}} \rightarrow y \in D^{\mathbf{D}}\}$
<code>restriction(U hasValue(v))</code>	$U : v$	$(U : v)^{\mathcal{I}} = \{x \mid \langle x, v^{\mathcal{I}} \rangle \in U^{\mathcal{I}}\}$
<code>restriction(U minCardinality(n))</code>	$\geq n U$	$(\geq n U)^{\mathcal{I}} = \{x \mid \#\{\langle y, x \rangle \in U^{\mathcal{I}}\} \geq n\}$
<code>restriction(U maxCardinality(n))</code>	$\leq n U$	$(\leq n U)^{\mathcal{I}} = \{x \mid \#\{\langle y, x \rangle \in U^{\mathcal{I}}\} \leq n\}$
Data Ranges (D)		
D (URI reference)	D	$D^{\mathbf{D}} \subseteq \Delta_{\mathbf{D}}^{\mathcal{I}}$
<code>oneOf($v_1 \dots$)</code>	$\{v_1, \dots\}$	$\{v_1, \dots\}^{\mathcal{I}} = \{v_1^{\mathcal{I}}, \dots\}$
Object Properties (R)		
R (URI reference)	R	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
	R^-	$(R^-)^{\mathcal{I}} = (R^{\mathcal{I}})^-$
Datatype Properties (U)		
U (URI reference)	U	$U^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta_{\mathbf{D}}^{\mathcal{I}}$
Individuals (o)		
o (URI reference)	o	$o^{\mathcal{I}} \in \Delta^{\mathcal{I}}$
Data Values (v)		
v (RDF literal)	v	$v^{\mathcal{I}} = v^{\mathbf{D}}$

Figure. B.2 OWL DL descriptions, data ranges, properties, individuals and data values

Appendix D:

Feature Pool Ontology in OWL Functional Syntax

```

Prefix (xsd:=<http://www.w3.org/2001/XMLSchema#>)
Prefix (owl2xml:=<http://www.w3.org/2006/12/owl2-xml#>)
Prefix (swrlb:=<http://www.w3.org/2003/11/swrlb#>)
Prefix (owl:=<http://www.w3.org/2002/07/owl#>)
Prefix (:=<http://wise.vub.ac.be/Members/lamia/variability/Feature_Assembly/FAM.owl#>)
Prefix (xml:=<http://www.w3.org/XML/1998/namespace>)
Prefix (rdf:=<http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
Prefix (swrl:=<http://www.w3.org/2003/11/swrl#>)
Prefix (rdfs:=<http://www.w3.org/2000/01/rdf-schema#>)

Ontology (<http://wise.vub.ac.be/Members/lamia/variability/Feature_Assembly/FAM.owl>

Declaration (Class (:Abstract_Feature))
EquivalentClasses (:Abstract_Feature ObjectIntersectionOf (:Abstract_Feature
ObjectAllValuesFrom (:Has_Option :Option_Feature)))
SubClassOf (:Abstract_Feature :Feature)
DisjointClasses (:Abstract_Feature :Concrete_Feature)
Declaration (Class (:Binding_Time))
SubClassOf (:Binding_Time owl:Thing)
DisjointClasses (:Binding_Time :Feature)
DisjointClasses (:Binding_Time :Keywords)
DisjointClasses (:Binding_Time :Perspective)
DisjointClasses (:Binding_Time :Product_Line)
DisjointClasses (:Binding_Time :Stakeholder)
DisjointClasses (:Binding_Time :Variability)
Declaration (Class (:Business_Analyst))
SubClassOf (:Business_Analyst :Stakeholder)
Declaration (Class (:Client))
SubClassOf (:Client :Stakeholder)
Declaration (Class (:Concrete_Feature))
EquivalentClasses (:Concrete_Feature ObjectIntersectionOf (:Concrete_Feature
ObjectAllValuesFrom (:Mandatory_Composition ObjectUnionOf (:Concrete_Feature
:Abstract_Feature)))
EquivalentClasses (:Concrete_Feature ObjectIntersectionOf (:Concrete_Feature
ObjectAllValuesFrom (:Optional_Composition ObjectUnionOf (:Concrete_Feature
:Abstract_Feature)))
SubClassOf (:Concrete_Feature :Feature)
DisjointClasses (:Concrete_Feature :Abstract_Feature)
Declaration (Class (:Developer))
SubClassOf (:Developer :Stakeholder)
Declaration (Class (:Domain_Expert))
SubClassOf (:Domain_Expert :Stakeholder)
Declaration (Class (:Feature))
EquivalentClasses (:Feature ObjectIntersectionOf (:Feature
ObjectAllValuesFrom (:has_Stakeholder :Stakeholder)))
EquivalentClasses (:Feature ObjectIntersectionOf (:Feature ObjectMaxCardinality(1
:has_Owner :Stakeholder)))
SubClassOf (:Feature owl:Thing)
DisjointClasses (:Feature :Binding_Time)
DisjointClasses (:Feature :Perspective)
DisjointClasses (:Feature :Stakeholder)
Declaration (Class (:Functional))
SubClassOf (:Functional :Perspective)
Declaration (Class (:Graphical_User_Interface))
SubClassOf (:Graphical_User_Interface :Perspective)

```

```

Declaration(Class(:Hardware_Interface))
SubClassOf(:Hardware_Interface :Perspective)
Declaration(Class(:Keywords))
SubClassOf(:Keywords owl:Thing)
DisjointClasses(:Keywords :Binding_Time)
DisjointClasses(:Keywords :Perspective)
Declaration(Class(:Localization))
SubClassOf(:Localization :Perspective)
Declaration(Class(:Marketing))
SubClassOf(:Marketing :Stakeholder)
Declaration(Class(:Modeller))
SubClassOf(:Modeller :Stakeholder)
Declaration(Class(:Non_Functional))
SubClassOf(:Non_Functional :Perspective)
Declaration(Class(:Option_Feature))
EquivalentClasses(:Option_Feature :Variant)
EquivalentClasses(:Option_Feature
                    ObjectIntersectionOf(:Option_Feature
ObjectAllValuesFrom(:Option_of :Abstract_Feature)))
SubClassOf(:Option_Feature :Feature)
Declaration(Class(:Persistent))
SubClassOf(:Persistent :Perspective)
Declaration(Class(:Perspective))
DisjointClasses(:Perspective :Binding_Time)
DisjointClasses(:Perspective :Feature)
DisjointClasses(:Perspective :Keywords)
DisjointClasses(:Perspective :Stakeholder)
Declaration(Class(:Product_Line))
SubClassOf(:Product_Line owl:Thing)
DisjointClasses(:Product_Line :Binding_Time)
Declaration(Class(:Project_Manager))
SubClassOf(:Project_Manager :Stakeholder)
Declaration(Class(:Sales))
SubClassOf(:Sales :Stakeholder)
Declaration(Class(:Stakeholder))
DisjointClasses(:Stakeholder :Binding_Time)
DisjointClasses(:Stakeholder :Feature)
DisjointClasses(:Stakeholder :Perspective)
Declaration(Class(:System))
SubClassOf(:System :Perspective)
Declaration(Class(:Task))
SubClassOf(:Task :Perspective)
Declaration(Class(:Testers))
SubClassOf(:Testers :Stakeholder)
Declaration(Class(:Variability))
DisjointClasses(:Variability :Binding_Time)
Declaration(Class(:Variant))
EquivalentClasses(:Variant :Option_Feature)
SubClassOf(:Variant :Variability)
Declaration(Class(:Variation_Point))
SubClassOf(:Variation_Point :Variability)
Declaration(Class(owl:Thing))
Declaration(ObjectProperty(:Belongs_to))
ObjectPropertyDomain(:Belongs_to :Feature)
ObjectPropertyRange(:Belongs_to :Perspective)
Declaration(ObjectProperty(:Composition))
ObjectPropertyDomain(:Composition :Concrete_Feature)
ObjectPropertyRange(:Composition ObjectUnionOf(:Concrete_Feature :Abstract_Feature))
Declaration(ObjectProperty(:Excludes))
SubObjectPropertyOf(:Excludes :FTFC)
SymmetricObjectProperty(:Excludes)
Declaration(ObjectProperty(:FTFC))
ObjectPropertyDomain(:FTFC :Feature)
ObjectPropertyRange(:FTFC :Feature)
Declaration(ObjectProperty(:Has_Option))
InverseObjectProperties(:Has_Option :Option_of)
ObjectPropertyDomain(:Has_Option :Abstract_Feature)
ObjectPropertyRange(:Has_Option :Option_Feature)
Declaration(ObjectProperty(:Mandatory_Composition))
SubObjectPropertyOf(:Mandatory_Composition :Composition)
ObjectPropertyDomain(:Mandatory_Composition :Concrete_Feature)
ObjectPropertyRange(:Mandatory_Composition
                    ObjectUnionOf(:Concrete_Feature
:Abstract_Feature))
Declaration(ObjectProperty(:Option_of))

```



```

InverseObjectProperties(:Has_Option :Option_of)
ObjectPropertyDomain(:Option_of :Option_Feature)
ObjectPropertyRange(:Option_of :Abstract_Feature)
Declaration(ObjectProperty(:Optional_Composition))
SubObjectPropertyOf(:Optional_Composition :Composition)
ObjectPropertyDomain(:Optional_Composition :Concrete_Feature)
ObjectPropertyRange(:Optional_Composition :Abstract_Feature) ObjectUnionOf(:Concrete_Feature
:Abstract_Feature))
Declaration(ObjectProperty(:Requires))
SubObjectPropertyOf(:Requires :FTFC)
TransitiveObjectProperty(:Requires)
Declaration(ObjectProperty(:Same))
SubObjectPropertyOf(:Same :FTFC)
SymmetricObjectProperty(:Same)
Declaration(ObjectProperty(:Used_in))
ObjectPropertyDomain(:Used_in :Feature)
ObjectPropertyRange(:Used_in :Product_Line)
Declaration(ObjectProperty(:Uses))
SubObjectPropertyOf(:Uses :FTFC)
TransitiveObjectProperty(:Uses)
Declaration(ObjectProperty(:has_Binding_Time))
ObjectPropertyDomain(:has_Binding_Time :Feature)
ObjectPropertyRange(:has_Binding_Time :Binding_Time)
Declaration(ObjectProperty(:has_Keyword))
SubObjectPropertyOf(:has_Keyword owl:topObjectProperty)
ObjectPropertyDomain(:has_Keyword :Feature)
ObjectPropertyDomain(:has_Keyword :Perspective)
ObjectPropertyDomain(:has_Keyword :Product_Line)
ObjectPropertyRange(:has_Keyword :Keywords)
Declaration(ObjectProperty(:has_Owner))
ObjectPropertyDomain(:has_Owner :Feature)
ObjectPropertyRange(:has_Owner :Stakeholder)
Declaration(ObjectProperty(:has_Perspective))
ObjectPropertyDomain(:has_Perspective :Product_Line)
ObjectPropertyRange(:has_Perspective :Perspective)
Declaration(ObjectProperty(:has_Stakeholder))
ObjectPropertyDomain(:has_Stakeholder :Feature)
ObjectPropertyRange(:has_Stakeholder :Stakeholder)
Declaration(ObjectProperty(owl:topObjectProperty))
Declaration(DataProperty(:Standalone))
DataPropertyDomain(:Standalone :Feature)
Declaration(DataProperty(:has_Description))
Declaration(DataProperty(:has_Label))
Declaration(DataProperty(:has_Rationale))
DataPropertyDomain(:has_Rationale :Feature)
DataPropertyRange(:has_Rationale xsd:string)
Declaration(NamedIndividual(:Analysis))
ClassAssertion(:Binding_Time :Analysis)
ClassAssertion(owl:Thing :Analysis)
Declaration(NamedIndividual(:Compilation))
ClassAssertion(:Binding_Time :Compilation)
ClassAssertion(owl:Thing :Compilation)
Declaration(NamedIndividual(:Design))
ClassAssertion(:Binding_Time :Design)
ClassAssertion(owl:Thing :Design)
Declaration(NamedIndividual(:Installation))
ClassAssertion(:Binding_Time :Installation)
ClassAssertion(owl:Thing :Installation)
Declaration(NamedIndividual(:RunTime))
ClassAssertion(:Binding_Time :RunTime)
ClassAssertion(owl:Thing :RunTime)
Declaration(NamedIndividual(:StartUp))
ClassAssertion(:Binding_Time :StartUp)
ClassAssertion(owl:Thing :StartUp)
Declaration(AnnotationProperty(:Dependency_Owner))
Declaration(AnnotationProperty(:Dependency_Reason))
Declaration(AnnotationProperty(:Enforced_Dependency))
DLSafeRule(Body(ObjectPropertyAtom(:Optional_Composition Variable(<urn:swrl#x>))
Variable(<urn:swrl#y>)) Head(ClassAtom(:Concrete_Feature ClassAtom(:Concrete_Feature
Variable(<urn:swrl#x>))) Head(ClassAtom(:Variation_Point Variable(<urn:swrl#x>))))))
DLSafeRule(Body(ObjectPropertyAtom(:Has_Option Variable(<urn:swrl#x>))
Variable(<urn:swrl#y>)) Head(ClassAtom(:Abstract_Feature ClassAtom(:Abstract_Feature
Variable(<urn:swrl#x>))) Head(ClassAtom(:Variation_Point Variable(<urn:swrl#x>))))))

```

```
DisjointClasses(:Perspective :Product_Line :Variability)
)
```