State of the Art: Integrated Management of Requirements in Model-Based Software Engineering

Christer Thörn

ISSN 1404-0018
Research Report 06:3

INGENJÖRSHÖGSKOLAN

HÖGSKOLAN I JÖNKÖPING

State of the Art: Integrated Management of Requirements in Model-Based Software Engineering

Christer Thörn

Information Engineering Research Group
Department of Electronic and Computer Engineering
School of Engineering, Jönköping university
Jönköping, SWEDEN

ISSN 1404-0018
Research Report 06:3

Abstract

This report describes the background and future of research concerning integrated management of requirements in model-based software engineering. The focus is on describing the relevant topics and existing theoretical backgrounds that form the basis for the research. The report describes the fundamental difficulties of requirements engineering for software projects, and proposes that the results and methods of models in software engineering can help leverage those problems. Taking inspiration from the advances of domain engineering, software product lines, and the application family concepts, the research topic described aims to facilitate requirements engineering utilizing modeling of commonality and variability. The primary vessel for this is feature models, describing the capabilities and potential of the set of products under consideration. The research will evolve the existing modeling notions such as meta-models and methodologies to suit the needs of requirements engineering.

Keywords

Model-Based Software Engineering, Requirements, Software Modeling, Domain Engineering, Feature models

Acknowledgement

Part of this work was financed by the Swedish Knowledge Foundation (KK-Stiftelsen), grant 2003/0241, project SEMCO.

Table of Contents

1 Introduction 1
1.1 Background 1
1.2 Methodology Description 2
1.3 Delimitations 2
1.4 Disposition 2
2 Requirements Engineering 3
2.1 Requirements Engineering Basics 3
2.1.1 Requirements Types 3
2.1.2 Problems in RE 4
2.2 Requirements Engineering Processes 5
2.2.1 Requirements Elicitation 6
2.2.2 Requirements Analysis 7
2.2.3 Requirements Documentation 8
2.2.4 Requirements Validation 8
2.2.5 Change Request Management 9
2.3 Requirements Engineering Techniques and Models 9
2.3.1 Elicitation Techniques 9
2.3.2 Requirements Models 10
3 Reuse and Modeling in Software Engineering 12
3.1 Software Reuse 12
3.1.1 Opportunistic vs. Managed Reuse 12
3.1.2 Application Families 14
3.1.3 Reuse-driven Software Engineering Business 14
3.2 Software Modeling 16
3.2.1 Modeling in Engineering 16
3.2.2 Model Use in Software Engineering 18
3.3 Our View of Model-Based Software Engineering 19
4 Approaches in Model-based Software Engineering 21
4.1 Domain Engineering 21
4.1.1 History 21
4.1.2 STARS and Generative Programming 22
4.1.3 Families 27
4.2 Software Product Lines 29
4.3 Model Driven Architecture (MDA) 32

5 Feature Models **34**
5.1 History 35
5.2 Definitions 35
5.2.1 Features 35
5.2.2 Feature Classification 37
5.2.3 Constraints and Relations 38
5.3 Feature Diagram Notations 40
5.3.1 FODA 40
5.3.2 Czarnecki and Eisenecker 42
5.3.3 Riebisch 42
5.3.4 Supplemental Information and Attributes 44
5.4 Methodologies and Uses of Feature Models 46
5.4.1 FODA/FORM 47
5.4.2 The Generative Programming Feature Finding Process 47
5.4.3 PuLSE-CaVE 48
6 Conclusions and Research Questions **50**
References **51**

1 Introduction

This report describes the theoretical foundations and relevant background topics for research in integrated management of requirements in model-based software engineering. It attempts to show what work has been done so far in the fields that are of concern for the research and where the main influences for the work to be conducted in the research project originate. This report summarizes and highlights important and interesting topics in requirements engineering, software engineering, domain analysis and feature models, which are all instruments of the research. The topics are quite wide-ranging and although the intent is to focus on the relevant portions, it is hard to discuss such huge topics as software engineering without putting it into context by providing an overview.

1.1 Background

The term ”software crisis” has been around for a long time and it seems that all solutions that have been suggested have introduced further more problems. As a result, one of the most disparate and interesting research areas today is software engineering, which tries to solve the problems that are facing software developers. For the intent of the report at hand we can focus on some particular statements of modern software intensive systems.

First, much of the problems that concern the difficulties of achieving success in software projects can be traced to the requirements engineering activities of the software development process. This emphasizes the importance of requirements engineering and means that there are considerable gains to be achieved if the requirements engineering activities can reach its goals of stating unambiguous and complete demands on the product. This research aims at providing support for the requirements engineering process.

Second, the use of models throughout the development process has proven to be a very useful approach in software engineering. As the software production industry has moved towards applying the engineering principles, models have become a prevalent and powerful tool to assist the construction of software. To reach the benefits of modeling, such as the ability to abstract, communicate and test designs before implementing them, are also of interest for this report.

Lastly, the importance of understanding the domain and potential of domain analysis has clear benefits and advantages. The premise of the research is to support the requirements engineering process through the use of models for domain understanding. By applying models to support the requirement engineering activities, the work hopes to facilitate avoiding the problems that are often associated to the requirements engineering.

1.2 Methodology Description

The basis for the contents of this report is an extensive literature study of the different topics treated in the report. The report brings up both background information as found in most textbook literature as well as recent findings and research.

1.3 Delimitations

As already mentioned this report is delimited in selecting only parts of the general topics discussed for detailed excursion. The selection is obviously based on what is deemed as relevant for the topic of the research. In particular, the focus of the project is on requirements management, which means that in the sections related to methodologies and process models, much of the material relating to other activities during software development are left out.

1.4 Disposition

The purpose of the report is to bring up the necessary theoretical grounds to appreciate the research questions and it does so in the following manner.

The next section (2) will describe the fundamentals of requirements engineering and the problems and techniques that are related to the requirements engineering activities. The following two sections discuss the general topic of model-based software engineering (3) and some specific approaches in model-based software engineering (4), respectively. Having covered the basics of requirements engineering and developing with models, with an understanding of the possibilities and problems associated with them, we present a particular model which has the prospect of being able to tie in the requirements engineering phases with the rest of the model-based development activities, namely feature models (5). Finally, we conclude the report with possible research questions and lay out the work needed to answer those questions (6).

2 Requirements Engineering

The requirements engineering (RE) activity is part of all processes for software development and aims at capturing the needs and demands that are posed on the system by the customer and other stakeholders. Understanding the requirements is crucial for producing good software, since most of the reasons why software projects are delayed, break budgets or fail involve some factor of requirements engineering [39]. Since the costs and the efforts associated with repairing the effects of faulty requirements management are potentially substantial, getting the requirements right from the start is important. As most software development processes would begin with customer or perceived market requirements, and the work during the rest of the development process is mandated by what the requirements handling activities produces, there is great potential for achieving benefits of managed reuse, as discussed in section 3.1, if one could extend the reusable assets to include requirements.

This section will cover the essential concepts in requirements engineering that are related to processes, techniques and models that are used in requirements engineering. Most of the material is gathered from [2, 32, 45] and [39]. Although there is much to be said about requirements engineering, we focus this excursion on the parts that are of more concern to the research topics at hand.

2.1 Requirements Engineering Basics

2.1.1 Requirements Types

Most people have an intuitive feeling for what a requirement is and what it embodies, but for formality it can be defined as a statement of system service or constraint. The requirements that are posed on a system by its stakeholders should provide an unambiguous and complete specification of what the system should be able to do and what it should result in. The requirements should feed the design process and provide the frames in which the developing organization can work. A requirement comes in several different forms and granularities:

1. General statements that broadly describe what the system under consideration should do. These requirements are more of descriptions of intentions rather than requests to build against.

2. Functional requirements, which describe what functionality and capability the system should have. These are the most common forms of requirements found in requirements specifications.

3. Implementation requirements, which specifies actual implementation constraints. These are discussed below.

4. Non-functional requirements, which describe quality factors that does not fall cleanly under functional requirements. These could involve for instance performance, reliability and usability. Factors like these are transient for the entire system and do not reflect any one part of the system.

5. Constraints and interfaces, which define how the system under consideration should interact and interface with the environment. More on constraints below.

The general idea on requirements is to describe the demands posed on the system by the stakeholders in an implementation independent way. Some constraints will unconditionally be imposed on the system, for instance by regulations and laws, standards, or internal or external policies. These constraints will limit the freedom of implementations and the space in which the designers of the system can look for solutions.

Although the holy grail of requirements engineering has always been to think of the system being developed as a black box, and formulate requirements as demands on how the black box is expected to interact with the surrounding and what effect it will have on the context it exist in compared to if it had not existed, it is in practice impossible to keep a water tight hull between requirements and design or implementation. The upside of this is that requirements that are practically unfeasible are discovered and weeded out at an early stage based on the experience of the developing organization. The downside is that it might place constraints on the development and hinder the selection of the most suitable solution to the problem. This means that while traditionally requirements engineering and design has been argued to be separate activities, a more pragmatic approach emerged where requirements engineering and design are seen as interlaced activities.

2.1.2 Problems in RE

We have already mentioned some shortcomings of individual requirements. Now we turn to the problems of the requirements engineering activity. Even though the requirements engineering activity has been a part of all development processes for systems or engineering products for more or less all time, there are still many problems related to it.

The most obvious problem is of course that the requirements do not end up reflecting the real needs of the stakeholders of the system. This is related to inconsistencies or incompleteness of the requirements, as well as misunderstandings between the stakeholders and the developers. We will discuss models in requirements engineering later on, but for the most part requirements are written using natural language. This means that although natural language can be very expressive concerning complex issues, requirements written in actual text risk not being sufficiently formalized and thus mean different things to different readers. It could also be the case that

the needs of the system are not fully understood by any of the involved parties. For instance, the stakeholders of a library system could probably be quite content with the identification of objects by ISBN, while not considering the fact that for instance CD-ROMs do not have ISBNs.

Most systems, bar the simplest, will have more than one stakeholder. The demands posed on the system by the stakeholders are most likely going to be divergent in some respects and sometimes incompatible. During requirements negotiations the stakeholders has to make compromises which in turn could effect other requirements for the product. Requirements interactions can cause complications that need to be resolved during requirements analysis. The interactions can cause unexpected effects and conflicts, which give rise to other requirements that emerge as a result of overall system behavior and functionality.

2.2 Requirements Engineering Processes

Over time many process models have been developed for requirements engineering. The process models are obviously tightly related to requirements engineering models and techniques, which are covered in the next subsection, so this subsection will describe the fundamental activities common to most, if not all, requirements processes. Processes, being a set of organized activities transforming inputs to outputs, of course aim at building on the experiences and best practices for performing some task. Although practice is supposed to make perfect, the requirements engineering track record indicate that there is still room for improvement.

The activities that make up the requirements engineering process are usually divided into elicitation, analysis, negotiation, documentation and validation, each activity building on the results from the previous. The result of the requirement engineering process is a requirements specification that can be used for design specifications. Each activity is explained below.

There are several variations of the requirements engineering process, many of them based on more generic process patterns that have also been applied to complete software development processes. The classical waterfall type of process, as seen in Figure 1, is of course an ideal and probably the most widely used and best understood process. This makes it easy to reason about and modify by implementing new functionality. Although ideally thought of as a one-way information flow, the activities in fact provide feedback to one another and several iterations of the activities has to be completed. Accordingly, most descriptions of this process model have been supplemented with some form of feedback loops between the activities.

A process that makes the iterative nature of activities in the process more explicit is the spiral model (of course popularized by the spiral model for software development by Boehm). Applied to the activities in requirements engineering, Figure 2

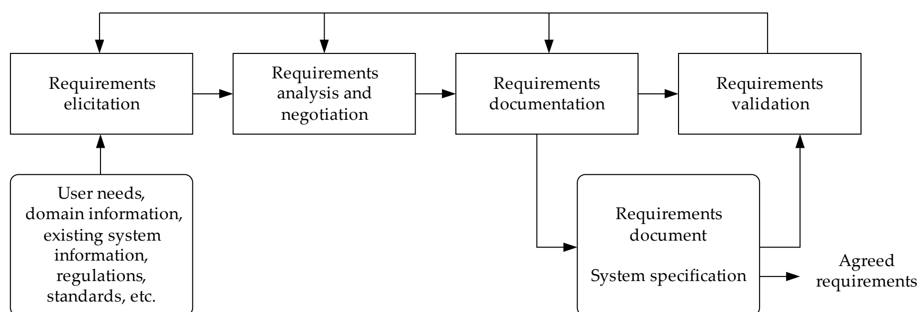

Figure 1: A fundamental process model for typical waterfall work flow in requirements engineering. Adapted from [32].

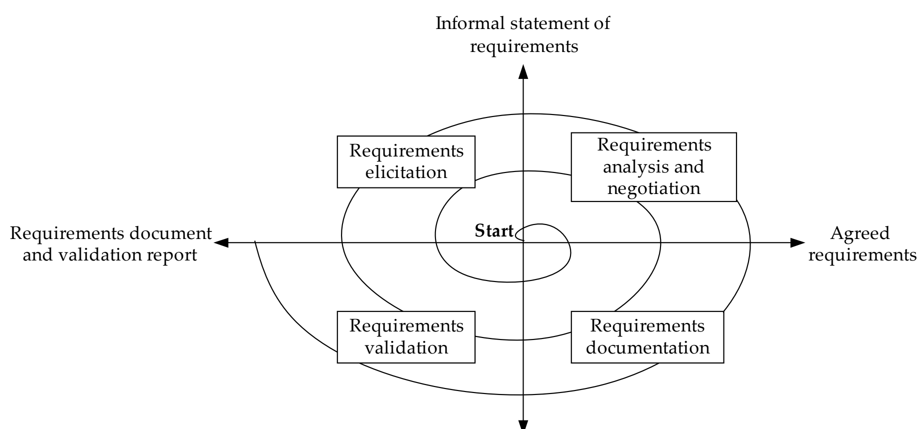

Figure 2: Illustration of the spiral model applied to requirements engineering. Adapted from [32].

shows how the iterations in the spiral gradually refine the results of the activities into a form that eventually is accepted as the final specification of requirements.

2.2.1 Requirements Elicitation

Requirements elicitation, also known as requirements acquisition or requirements discovery, finds the actual requirements in cooperation with the stakeholders of the project. [32] points out that the elicitation is not a ”fishing” for requirements, but rather a process in it self to discover the needs of the stakeholders. The elicitation is often complicated by the fact that the stakeholders approach the problems that are to be solved from different perspectives, with little understanding of other viewpoints. It is therefore necessary to treat the elicitation systematically. Figure 3 shows four dimensions that make up such an approach to elicitation:

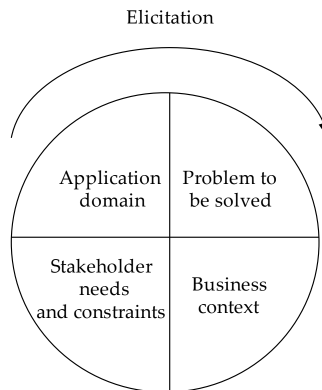

Figure 3: Dimensions in requirements elicitation. Adapted from [32].

- Understanding the domain means to understand the general area and context in which the system under development is supposed to act. For example, in order to build software systems concerning automotives, it is necessary with background knowledge of automotives in general.
- Problem understanding involves realizing the details of the problem to be solved, for instance, details of sensors and actuators to be used with the software. This component of elicitation extends the general domain knowledge.
- Business context is the understanding of the business goals that the system is intended to contribute towards and how the system will affect the organization and the interacting parts of the business.
- Once the domain, problem and context are understood, comes the understanding of the particular needs that the stakeholders want the system to fulfill and provide solutions for. Essentially it means to understand the support the stakeholders expect from the system and what role it will play in existing settings of the stakeholder.

2.2.2 Requirements Analysis

Some requirements analysis is carried out already during elicitation, as some problems with the requirements might be immediately recognized and resolved right at the source. Even so, the requirements have to be analyzed in detail in order to decide which ones that should be accepted and brought into production. Inevitably, there will be conflicting requirements that have to be resolved against each other, which leads to this activity sometimes being called requirements negotiation.

Some reasons for conflicting requirements have already been mentioned as stakeholders having different viewpoints and expectations. Other reasons could be that the sources where details about requirements are acquired, such as documentation,

standards etc., are incomplete or not sufficiently detailed to provide the information needed to make it into a requirement. There could also be the case that the budget puts constraints on the number and type of requirements that can be put into the project.

2.2.3 Requirements Documentation

The requirements agreed upon need to be documented at an appropriate level of detail in order to be used during the rest of the development activities. The goal is to make sure that the documentation of the requirements is as unambiguous and understandable as possible, so that all stakeholders can understand it. This has traditionally meant that the requirements are written using natural language in plain text. While this gives a lot of freedom to write expressively, it also leads to large text corpuses that are hard to penetrate and hard to maintain during the development activities.

The need for tool support for requirements management was realized early on and several systems were developed for writing, organizing and searching the requirements of a project. Most of these systems where focused on handling the tracing of requirements and in a sense was not much more than quite fancy text editors with the ability to add some meta-data. Apart from the textual descriptions, there could be some diagrams, figures and sample models to explain or relate requirements, but these were in effect mere extensions of the textual requirements descriptions. In the next subsection 2.3, we will describe other ways to describe requirements and how to model the requirements.

2.2.4 Requirements Validation

The final activity of a requirements engineering process is to validate the requirements that goes into the documentation in order to ensure that the documentation can be used as a specification to accordingly build the system by. The activity is supposed to discover more or less the same type of conflicts and problems as in the analysis phase. The validation should try to ensure that all the documentation can be agreed on and that the produced requirements material is usable as a common ground for all further development activities.

Once the validation has ensured that the requirements documentation is consistent and complete, it can, in theory, be used as basis for the design and implementation of the system. In practice, this is however not the end of the requirements process. As the validation detects errors and problems with the specifications, new requirements emerge or display interdependent behavior, which means another run through the process with elicitation, analysis, negotiations and validation has to be performed.

2.2.5 Change Request Management

Another more direct reason that there is no definite end to the requirements engineering process is that the requirements posed on the system are likely to change over time for reasons external to the system itself. Due to changes in market, organization or environment, changes to the system will be necessary. For example, new legislation or standards could come into effect during the time that a project is being developed.

These changes in needs from the system make existing requirements obsolete or insufficient to reflect the demands that are posed on the system. Such events are often referred to as change requests, and suffer the burden mentioned previously of being hard and potentially expensive to remedy. In the best case, the change request only affects a single requirement, or isolated set of requirements. More probably, however, a change request would affect many parts of the product and requirements dependencies would cause larger changes to be necessary.

Ways to mitigate the risk for change requests are to try anticipating future needs from the stakeholders' side. By having the product fulfill more than what was originally requested in the requirements, the product will be more flexible. Of course, this can place unreasonable constraints and demands on the product and is often not a serious option. The other way is of course to ensure that the original requirements are as complete as possible and make sure that the stakeholders in the project are made aware of the needs and capabilities at an early stage during requirements engineering.

2.3 Requirements Engineering Techniques and Models

In this subsection we will consider some of the approaches for requirements engineering that have emerged as a complement or substitute for the textual descriptions already mentioned. Most of the approaches are based on a model or a viewpoint of the requirements elicitation process, that governs the technique used during the requirements engineering activity. Examples of useful views for requirements processes are object-orientation, where the principal viewpoint of the system is that all components of it are objects with data and operations, or scenarios where the requirements engineer and stakeholders start from examples of use-cases for the system.

2.3.1 Elicitation Techniques

There are many techniques used for finding the requirements from the stakeholders, of which we will mention a few here that are relevant. The principal method is to elicit requirements through interviews with the stakeholders. The interviews are meant to have the stakeholder present the needs and demands posed on the system, while the requirements engineer record the needs. The interview can either be struc-

tured so that the stakeholder answers a set of questions prepared in advance by the requirements engineer, or it can be of a more free form. Usually, an interview will tend to contain elements of both structured and unstructured methods. In either case, one thing that can leverage the results is to try to employ indirect reuse of requirements. By letting the stakeholder comment and criticize existing requirements, there is a baseline to relate the stakeholder's specific requirements to.

By observing the stakeholders in their ordinary work situations, requirements that would otherwise be hard to elicit through other means can be found. Observation is typically useful for finding things out about a stakeholder's situation that is hard to explain in a structured interview or during some other elicitation technique. Observation in conjunction with open-ended interviews where the stakeholder explains the tasks can be an important supplement to other techniques. One can also find important information in process models, enterprise descriptions and other documentation which already exist in the organization.

Scenarios are stories of interaction with the system. Letting stakeholders relate to examples of how the system will be used by them can be an efficient way of adding more detail to an outlined requirements specification. The scenario lets the stakeholder describe the problems and limitations that is experienced while working through the scenario. Scenarios can be written and carried out in different ways, but normally include descriptions of the starting conditions, flow of events, exceptions and the state of the system after the scenario has been completed. The next section will cover use-cases which are modeling constructs related to the use of scenarios.

2.3.2 Requirements Models

Most of the modeling approaches that exist, such as (Unified Modeling Language) UML, (Unified Process)UP, SADT, etc., provide a host of different solutions and notations for modeling parts of the system that essentially belongs to the requirements elicitation phase. Most of them generally divide into conceptual models, flow models, and scenario models. Conceptual models are often conjured up when an object-oriented approach is taken to requirements elicitation. The models often describe key objects and entities in the system, see Figure 4. Flow models, as seen in Figure 5, describe how information and data moves and communicates between elements and stakeholders.

One of the most commonly known scenario models is the Use-Case-model, as devised by Jacobsen and now often used by methodologies to support requirements elicitation. Use-cases, as depicted in Figure 6, are often used with scenario-based elicitation techniques and to provide an overview of what stakeholders that interact with different parts of the system. The model shows the stakeholders, the scenarios and how they relate to each other using various stereotyped extensions.

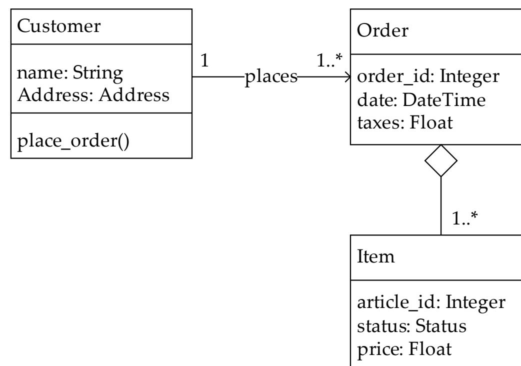

Figure 4: A sample of a conceptual model.

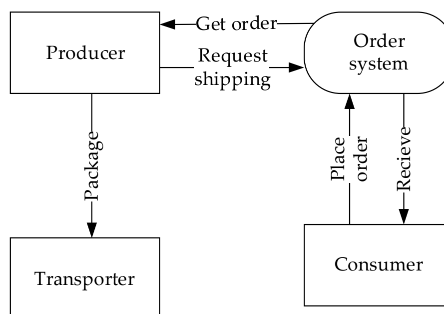

Figure 5: A sample of a data flow model.

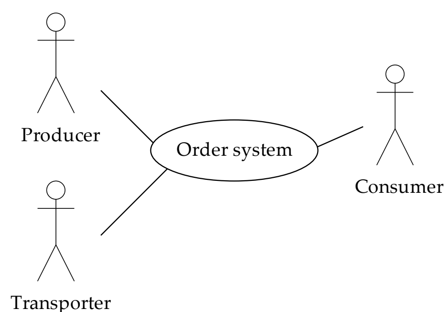

Figure 6: A sample of a use-case model.

3 Reuse and Modeling in Software Engineering

This section introduces the background of relevant topics in software engineering, specifically the notions of software reuse, illustrated with a sketch of a reuse-oriented methodology, and the use of models in software engineering. In conclusion we present our view on model-based software engineering, based on the discussed topics. Much of the material in this section is adapted from [28, 39, 44]. The topics in this section are complemented and revisited in section 4, which elaborate the subject as it describes specific approaches to model-based software engineering (MBSE).

3.1 Software Reuse

As the demands placed on software continuously increase and the challenges of producing software of high quality and high complexity at a low price increase, the processes and methods for producing software must evolve to meet these demands. The essence of the desired quality sought for in software is cost efficiency. Cost efficiency has historically been a hot topic in all engineering disciplines and has traditionally been solved in much the same way. The two classical approaches to achieve cost-efficiency are automation and reuse. Automation is usually made part of an engineering discipline and integrated into the work quite early. In software engineering, examples of basic automation are for instance build environments, code generators and CASE-tools. More efficient tool support and development of more efficient methodologies and work patterns lead to improved productivity and thus achieve the goal to a certain degree.

3.1.1 Opportunistic vs. Managed Reuse

There are two principal ways to employ software reuse, which we call opportunistic reuse and managed reuse. Opportunistic reuse is the intuitive form of reuse which is prevalent in all forms of engineering on some level. It basically means that individual developers find themselves in a situation where they based on their experience notice similarities in the software and design, prompting them to reuse fragments of various sizes from previous development efforts. This form of reuse is ad-hoc and un-controlled in that the developers are not coordinating the reuse efforts in the organization. This form of reuse commonly includes design elements and program code and is generally centered on implementation work products. While the initial attempts to reuse software material during development was comprised of code modules and libraries containing commonly used implementations, larger and more complex systems demand greater levels of reuse.

The benefits of reuse are similar throughout all development and engineering domains. Reusing tried and tested solutions increases the quality of the produced goods

and the effort that can be saved by employing reuse leads to faster time-to-market and economies of scale in that increased reuse of artifacts results in more products per artifact and thus better spent effort. The use of inter-exchangeable parts based on a common framework means that the organization can respond faster to changes in market and provide more options to customers, increasing the agility of the products. The prospect of attaining these benefits fuels the interest in what we refer to as managed reuse. Rather than employing an ad-hoc form of reuse and hoping that new software development projects will be able to use previously produced material, the projects and the produced material is planned and managed from the start to maximize the potential of reusing past solutions. The material that is produced in the projects should also be designed and implemented in such way that it will be possible to reuse in future projects.

Managed reuse is not a recent paradigm, but originates like many other software engineering concepts from the end of the 1960s. The level of implementation of the concepts has however varied over time. The idea of managed reuse can obviously be applied on different levels depending on the ambition of the organization to achieve reuse. The reuse efforts could continue to be focused on code and result in modules and libraries of commonly used functionality throughout the development projects in the organization, which enables efficient use of specialist competence. Moving up the ladder of abstraction, the organization could construct components for reuse in projects, consisting of self-contained modules of data and functionality. Once the components are in place, one creates a framework to plug the component into. The component-based reuse was made popular, and readily possible to implement, with the break-through of object-oriented programming. Today, many component-based frameworks and methodologies exist, such as CORBA, COM, J2EE, etc. (For more on this subject see for instance [47].) Moving further, we extend the reuse beyond implementation to involve also requirements, design, and test, reusing all aspects of the project. Managed reuse has demonstrated better results in efficiency than tool and process improvement, although the results of the latter are not negible. [28]

A fundamental difference to the opportunistic, ad-hoc form of reuse mentioned previously is that the managed reuse is set to take place on an organizational level, rather than on the individual level. The reuse-policy is organization-wide and supported throughout all the projects that are undertaken. The methodologies that concern managed reuse, all stress the importance of enforcing the reuse, so that the organization not only encourages reuse, but also make sure it is performed at any point where it is possible. The managerial aspect is important to counter some of the problems which could occur in a reuse-oriented organization, most notably the not-invented-here-syndrome and the lure to implement a quick, custom solution to a problem rather than taking the extra effort to produce a reusable solution. An important aspect of this is the matter of funding for the development projects. Generally, most efforts into enabling reuse throughout an organization has shown that most problems and issues that have caused obstacles, are usually not technical in na-

ture, but rather a matter of insufficient support from development processes, lack of commitment from the management, and continued focus on single-system thinking.

3.1.2 Application Families

A successful and proven approach to software reuse is the concept of application families. The idea of system families was proposed in [38] in the mid 1970s where analysis of a set of similar systems would render common extractable and reusable assets. The core of the family of products would be used in all instantiations and the development of a new product would consist of adapting reusable components and possibly develop a few new components with functionality that can not be accommodated by the existing components.

[44] gives three examples of specializations that can be made when developing a new product:

1. Platform specialization, where different versions of the product runs on different platforms. See MDA, section 4.3 for more on this.
2. Configuration specialization, where different versions of the product are adapted to work in different environments and interact against different interfaces.
3. Functional specialization, where the products are developed to meet different customer requirements and demands.

The activities of developing a new family member are illustrated in Figure 7. Although one strives to achieve as much reuse as possible it is almost unavoidable that there will be some amount of custom development necessary to supply the customer with a product that meets their requirements. This brings about one of the more important issues of family-oriented reuse. As the need arise for custom development, it might seem tempting to customize existing frameworks, components and other reusable assets so that they fit into the current context, but by doing so break the compatibility with the rest of the family that the asset is reused in. This leads to the development of several distinct products which has to be maintained separately, and goes against the whole idea of families that reuse common components. Again, this is mainly an issue of management, rather than technology.

More on these issues in section 4.

3.1.3 Reuse-driven Software Engineering Business

An approach and methodology in the field of managed reuse is the Reuse-driven Software Engineering Business (RSEB) [28]. It endorses reuse using modeling activities,

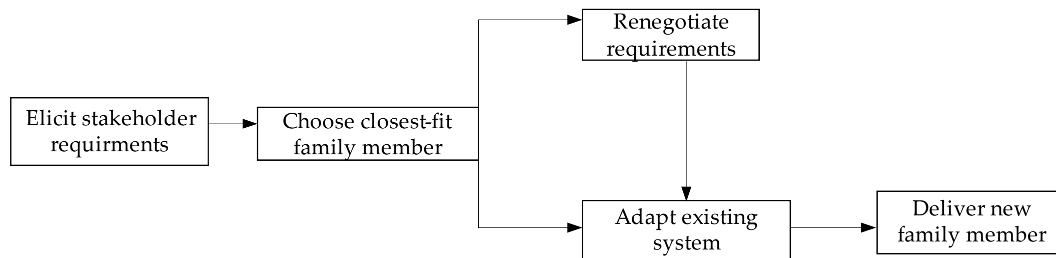

Figure 7: The steps involved in developing a new family member. Adapted from [44].

object-orientation and component-based architectures. The principal point made in RSEB, is the transition of an organization from being oriented towards single-system projects, to an organization that has as its business strategy to produce many-of-a-kind systems, with extensive reuse. RSEB is a development methodology and business philosophy, at the same time introducing development processes and technical solutions, while also discussing organizational structures and issues related to business management. Many other efforts similar to RSEB follow this same pattern in attempting to provide an all-encompassing solution for solving the problems of software engineering.

Like many all other such methodologies, RSEB emphasizes that in order to maximize the return on reuse in software engineering, most organizations must undergo changes in structure and other managerial aspects. RSEB points out the need of systematic and incremental changes in an organization to make the transition to a reusing organization. Figure 8 illustrates some typical steps and increments that an organization can go through to reach this state. RSEB uses the term ”business” to give weight to the fact that it is important that all aspects of the organization is oriented towards reuse, from corporate strategy and vision, to financing and project management, down to the fundamental parts and tasks in the development process used in the organization concerning technical details. Reaching this level of sophisticated reuse requires a systematic and decisive approach. The major issue is of course that the usual operations and projects must continue to progress, regardless of the changes in the organization. Many reasons for failure to employ a systematic reuse is due to the organization not following through on the transition plans, because of the high toll that is required. In fact many organizations that have successfully implemented reuse organizations, have been pressured into it by necessity rather than choice. [8]

RSEB has a significant element of modeling in it and defines formally software engineering as a process of building several related models. RSEB use, like many other methodologies that produce models, the unified modeling language (UML) to produce models. (Hardly surprising as one of the authors is the inventor of much of what would become incorporated in UML over the years.) Since RSEB subscribes to an object-oriented, component-based development methodology, the use of UML is particularly suitable as it provides most of the modeling constructs necessary for

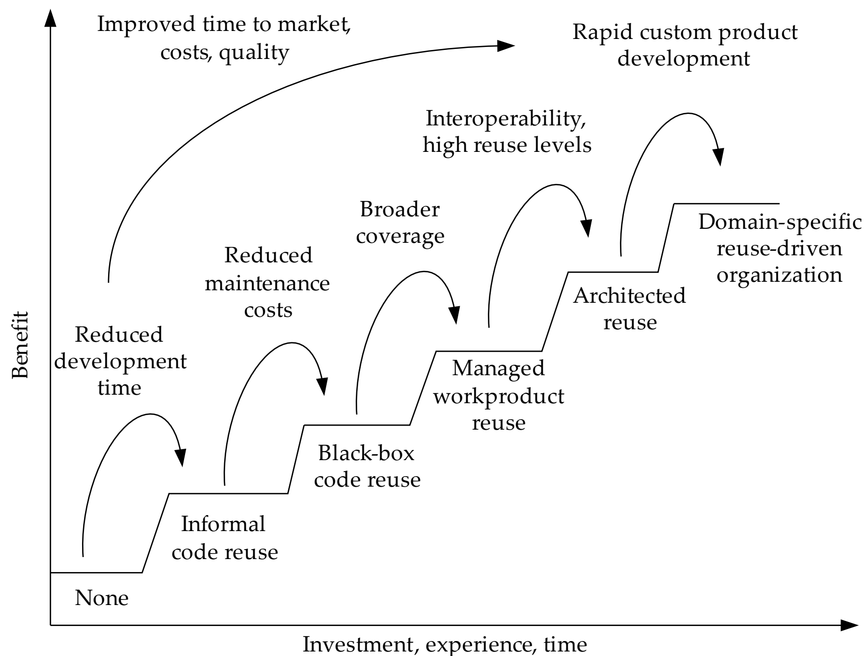

Figure 8: The incremental adoption of reuse according to RSEB. Adapted from [28]

this. More on UML is found in section 4.3 where it is discussed along with Model-Driven Architectures. The RSEB was followed by FeatuRSEB [21] which includes feature modeling activities. More on feature modeling is found in section 5. The next subsection will further discuss the use of software modeling.

3.2 Software Modeling

This subsection discusses the premise of which models exist and are being used, starting with basic model theory, continuing with particularities and peculiarities concerning the use of models in software engineering and concluding with remarks on the modeling topic that are of special relevance to the topic of this report.

3.2.1 Modeling in Engineering

A model is in its most abstract sense described as a conceptual representation of some process or interactions, or a theoretical construct of phenomenon. In the engineering principles and in the natural sciences, we usually find models as being abstracted, simplified and idealized constructions of, possibly non-existing, real-world concepts or processes. In [36] it is pointed out that a strength of models is the power of abstraction, for instance letting a toy-car model not represent just one single car, but all cars of that particular make, essentially modeling a class of objects.

Models in engineering come in many domain specific forms. Construction engineering use building blueprints, mechanical engineering use plans and electronics use circuit schematics to describe the solutions within their line of work. In all cases the purpose of using the models is similar. They assist in the development and manufacturing of products or artifacts by providing information about the consequences of building the artifacts before they are realized [36]. Modeling techniques also support the planning and structuring of the development activities [20].

The benefits and use we get from using models in development and engineering work are for example the possibility of having a formal and common understanding of concepts, processes and behaviors concerning a particular subset of our world. This understanding can be reached between all the stakeholders of the modeled concepts. Models allow us to represent problems and solutions to problems in an implementation-independent way using abstraction and simplification. They allow the developers to simulate and evaluate how design decisions will affect the artifact without having to take the risks that might be associated with a full-scale experiment. They also collect and provide information that can be stored and retrieved later at a point where it might be un-practical to acquire the information through interactions with the real artifact. Obviously, models as most people know them also provide an alternative way of presenting essentials of artifacts, most notably through visualization. A human tends to have vastly greater understanding for a visual representation of a modeled subject, compared to for instance tabular data.

According to [46] a model should meet three criteria:

- Mapping, meaning that there is some original object or phenomenon that the model is a representation of.
- Reduction, meaning that the model does not reflect all properties and attributes of the original. However, the model has to mirror some of the attributes of the original.
- Pragmatic, meaning that the model serves as a meaningful and usable replacement for the original in some context or for some purpose.

The mapping criterion does not imply existence of the original. For instance, a cost estimation or project plan is speculative concerning the actual costs or time planning. The reduction of the original to the model means that many attributes of the original are waived in the model. On the other hand, other attributes are added to the modeled object as abundant attributes that are not reflected in the real object [36,46] (Figure 9). In this vein we could consider the descriptions true of the original or world as DW, and the descriptions true of the software or the product as DM, in which case the modeling relationship are those that are in their intersection (Figure 10) [27].

Models come in two principal flavors, descriptive models which mirror an existing original, or prescriptive models which can act as an instruction for constructing an

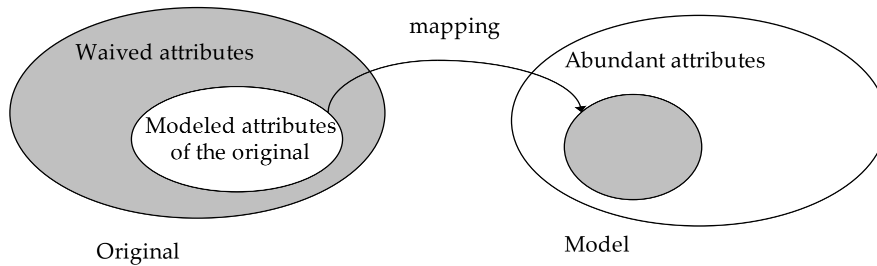

Figure 9: The relation between original and model according to Stachowiak. Adapted from [36].

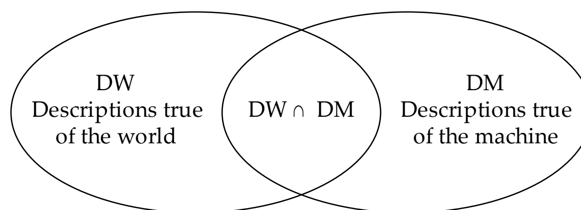

Figure 10: The relationship between descriptions of the world and the machine, forming the model. Adapted from [27]

original according to the description in the model. Descriptive models do not have to be created after the original has been created, since a prognostic model, such as a cost estimation are descriptive as they can not (directly) influence the original. [36, 46]

3.2.2 Model Use in Software Engineering

In software engineering, most models are prescriptive. [36] lists process models, information flow models, design models, user interaction models, models of principles such as patterns and process maturity models. Obviously each of these cases can be refined and further specified into the various models and constructs that exist within each of the many modeling paradigms and notations that exist.

The position of models in software engineering is quite strong, and rightfully so. In a sense software itself is very often a model of the world. So unlike traditional engineering disciplines, the artifacts that are resulting from software engineering are themselves models of complex processes, relations and objects. Due to the inherit complexity of the real world, which software models, software engineers use models to achieve reduction of complexity and abstraction of concepts, both of which are necessary to reach sufficient succinctness to be able to communicate the ideas set forth in the model to the stakeholders of a software development project.

Over time several modeling paradigms have come up, for instance the relational database model which is quite old and established, and the more recently there has

been a lot of interest in formal models for verification, which has not caught on quite as well. The most significant and influential advent in software modeling must however be object-orientation, which has by now become more or less ubiquitous and is considered the standard way of modeling and implementing software intensive enterprise systems. However, it is worth noting that even though almost every tool suite, vendor and software house more or less expects and presupposes object-orientation, the world is in fact not object-oriented. Although many of the modeling paradigms are very powerful, some more than others, and applicable in many situations for modeling the problems and solutions, there will always be the risk that the model chosen is inadequate for a correct formalization of the context it is supposed to describe. Although the UML is intended to be a general purpose language that can be extended to fit the bill for most purposes, strictest adherence to a particular modeling notation is not always the best guideline.

See the section 2.3.2 on requirements engineering for details on models used for requirements engineering.

3.3 Our View of Model-Based Software Engineering

There are many terms in the research community concerning the use of models in development and engineering, not only with regard to software, but also systems in general. Examples are Model Based Development (MBD), Model Driven Development (MDD), Model Driven Engineering (MDE) and other permutations in the same vein. All of them refer to development of systems using abstract representations with predefined syntax and semantics, supported by tools. [33]

In practice, the above terms mean more or less the same, namely leveraging the use of models to produce the products. We use the term MBSE for describing the work of this kind in our domain.

The aim of model-based software engineering is the continuous use of models throughout all stages of the software development and integrating them into a coherent workflow. Through a formal understanding of the organization and structure of the solutions used in the software, chances of achieving higher quality and greater levels of reuse are improved. Models constitute an abstracted representation of the solutions employed in the organization and its problems. Also, by using models a level of abstraction can be reached that lets the often very complex workings of software be communicated in a platform-independent way to stakeholders without technical insight.

Just like other industries have applied product line thinking and reuse for a long time, the software industry is trying to create ways of producing new software based on components and assemblies already available. When assembling software from

these assets it is of course imperative to have an adequate representation of them in some convenient format, which is where models come in.

As mentioned in the previous section, the choice of modeling notation and constructs should be governed by the application of the model, rather than to provide strict adherence to some format that does not serve the goal of the modeling activity. The quality of the model is paramount in this aspect and the ability to assess and ascertain the quality, benefit and usability of the model is very important in our view. Where traditional engineering disciplines have asked themselves if the model is a good predictor of the physical artifact or original, and what the assumptions, implicit or through simplifications, of the model mean to the behavior and impact of the artifact, we must find similar ways of valuing the quality of our modeling activities and produced results. [19]

4 Approaches in Model-based Software Engineering

This section will discuss some of the approaches to software engineering and software production that are relevant, given our view of model-based software engineering. The common trait of the methodologies brought up here, is that they all contain elements of software modeling, managed reuse and application family thinking, aiming to produce software intensive systems using these principles. This makes them influential on our work and form the history and state of practice to our work. The order in which they are presented here is in order of generality, more or less chronological order and in order of relevance, all of which happens to coincide. Whereas the methodologies tend to be quite elaborate, this summary will bring out the key points that form something of a core to all of them.

4.1 Domain Engineering

Domain engineering aims at facilitating organized software reuse by modeling the accumulated knowledge and capabilities within the business area of an organization, commonly referred to as the business domain. This can be translated to the software systems used in an organization by stating which tasks, data and operations that the system concerns. Several authors lists various definitions of the term domain, but the gist of all of them is that the domain encompasses a limited and scoped part of the world and the concepts contained within that domain.

Domain engineering is tightly connected with the concepts of software reuse. The intention of modeling a domain is obviously that as an organization constructs systems and conducts it business, it gathers experience and know-how. As most systems constructed in the organization are likely to share technical characteristics and are designed to meet similar requirements, it is likely that the organization can benefit from the acquired knowledge as subsequent systems are constructed. The domain model is an understanding of the domain, organized to take advantage of the experience and working with resources proven to be of importance, and domain engineering is a systematic approach to reach a state where it is possible to utilize these resources and assets.

4.1.1 History

The idea of domain analysis was introduced in the work conducted on program families in the mid 1970's. The term domain analysis was coined in 1980 by Neighbors [37] and described as ”the activity of identifying the objects and operations of a class of similar systems in a particular problem domain”. Over time several

methods and ways to perform domain analysis were developed, among the more notable Feature-Oriented Domain Analysis [31] and Organization Domain Modeling (ODM) [42]. A lot of work in the field of domain engineering and domain-based reuse of software was conducted in the research programs sponsored by the US Department of Defense, called Software Technology for Adaptable Reliable Systems (STARS) [10] and subprojects such as Central Archive for Reusable Defense Software (CARDS) [1]. These research programs spawned several variations and directions, among those the Software Engineering Institute's (SEI) Software Product Lines (SPL) [8], ODM and several other methodologies and guidelines for domain analysis. Since STARS was made up of several sub projects and carried out in cooperation with large industrial organizations for an extensive period of time, there is no single, coherent report of all the efforts that were the result of the program, nor would it make sense to try to squeeze all the work into some singular account. Instead we will settle for extracting the essentials in this report regarding the work done under STARS.

Neighbors express the key idea of domain oriented reuse in that ”it seems that the key to reusable software is to reuse analysis and design; not code.” The original idea of what to be captured in the domain model resulting from the domain analysis was refined and revised with the methodologies developed. Where the basic idea of ”objects and operations” from Neighbors remain, the advent of more advanced modeling tools, object-oriented modeling and other modeling methodologies meant that the domain model could be equipped with more advanced constructs, such as use-cases, feature models and concept models like class diagrams etc. [13]

Domain analysis as such was later made part of domain engineering and was integrated as part of a larger framework for working with model-based software engineering. The terminology has varied over time and between authors, but in modern literature and research, most have convened on the notion that domain engineering encompasses domain analysis, domain design and domain implementation. The word domain in the names of the phases, signals the intention on creating assets that are reusable for a system family.

4.1.2 STARS and Generative Programming

Building the domain knowledge repository and utilizing it are commonly referred to as domain engineering and application engineering respectively. The ideas that were set forth in STARS are reflected in how Software Product Lines [8] and Generative Programming [12] view the software reuse process.

Domain engineering are the activities that produce the reusable assets to be used during the activities that make up application engineering. Domain engineering is thus the activities that develop for reuse and application engineering is the activities that develop with reuse.

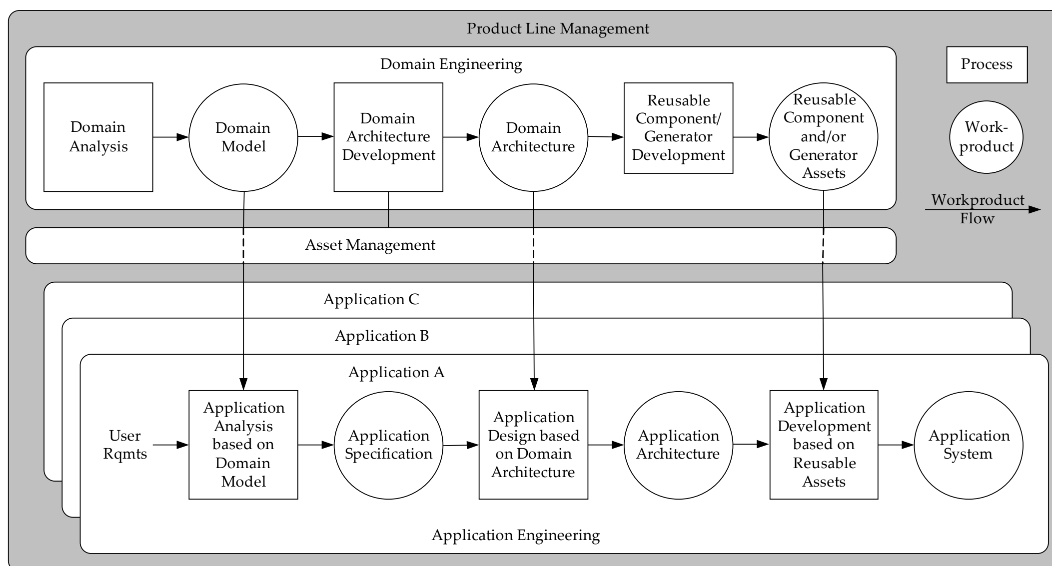

Figure 11: The Domain/Application engineering dual life-cycle. Adapted from [9].

The Domain/Application Engineering dual life-cycle model emerging from STARS, as seen in Figure 11, was adopted and adapted in most of the contemporary and later methodologies concerning domain engineering and organized software reuse. In this model for software development, the focus of the developers changes from a single system view on applications, to production and management of a family of applications. The model shows the intention of the two phases producing reusable assets and applications assembled from the assets. However, it does not display the iterative nature and the managerial issues needed to make an organization working in this manner function.

Generative Programming subscribes to the idea of having two parallel activities (see Figure 12) to collect, construct and organize the domain knowledge in an activity called domain engineering, and developing new products based on the reusable assets in an activity called application engineering. Since this is a commonly held and influential view in domain engineering originating from the SEI efforts, we use this section to introduce a baseline for the contents of the phases in domain engineering methods. This section is in large adapted from [12] and [13].

While conventional software engineering aims at satisfying the requirements for single systems, domain engineering produces assets usable to satisfy the requirements of a family of systems. [12] refers to areas organized around particular classes of systems as vertical domains, and parts of systems organized by functionality are referred to as horizontal domains. Domain engineering along a horizontal domain would result in reusable software components for user interfaces, communications etc., that is shared across several products in the software family. Domain engineering along a vertical domain would result in frameworks that could be instantiated to

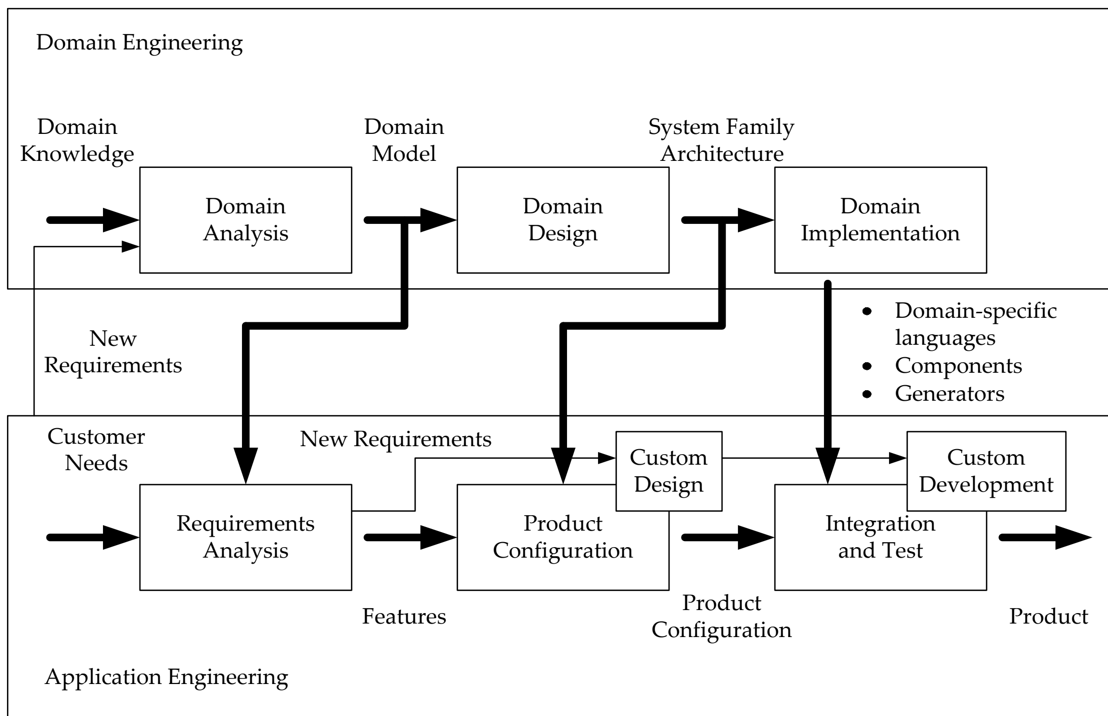

Figure 12: The workflows domain engineering at the top and application engineering at the bottom and the exchange of artifacts between them. Adapted from Generative Programming [12].

produce any system in the domain. The modeling of vertical domains should be done using several horizontal domains.

The Domain Engineering workflow

The domain engineering workflow consists of three major phases, domain analysis, domain design and domain implementation, each described below.

The domain analysis is the fundamental activity in domain-based software reuse and is typically the activity that initiates the whole process of adapting an organization to domain-based reuse. It is also the activity that has received most attention from researchers over the years. The domain analysis corresponds to the activities that would be carried out for any single system or software, but broadening the focus to a family of systems instead.

The first part of domain analysis is domain scoping. During this activity, the focus, boundaries and limits of the domain to be modeled are defined appropriately. While not making the scope to wide, which would reduce the chances of the organization being viable and able to successfully conduct their business, the scope does have to accommodate the potential of the domain in the future. It is important to use a scope that allows for sufficient flexibility in the products that are to result from the

development, but not let the scope of the domain stray so that the core assets can not accommodate the products. This would lead to a return to the classical development of one product at a time and one would loose the benefits that one hopes to achieve through software reuse. The scope should also identify the stakeholders and their interests that influence the domain. The stakeholders of the domain include managers, developers, investors, customers, end-users and so forth. It is argued that the delimitation of the domain is in fact the range of interests that the stakeholders have. The scope evolves as changes in market, organization and environment comes about.

Scope is determined on the grounds of marketing reasons and on technical basis. While a set of products might together satisfy a particular market segment, there could as well be a sensible set of products sharing technical characteristics that would make for a good software family. The term product family traditionally refers to a group of products that can be built from a common set of assets, based on technical similarity, whereas the term product line refers to a group of products sharing a common set of features that satisfy a need, based on marketing strategy. The distinction between product lines and product families is quite blurry and the terms are sometimes used interchangeably in literature.

The second part of domain analysis is the domain modeling, in which the domain model(s) are produced. The domain model is the explicit representation of the properties, semantics and dependencies of the concepts in the domain. It would typically be a set of different components and models, each describing one or more aspects of the system from a domain modeling perspective. Rather than all domain knowledge to be contained in one single model using a particular modeling language and notation, the strengths of a variety of modeling languages models can be utilized. The following components of a domain model are listed by Generative Programming [12]:

Domain definition Defines the scope of the domain in order to determine which systems that are encompassed by the domain and the rationale for inclusion and exclusion of systems for the domain.

Domain lexicon A taxonomy defining the domain vocabulary as it is understood by the practitioners in the domain.

Concept models Various models used to describe the concepts in the domain formally. This means models such as class diagrams, interaction diagrams, etc. Apart from formal models, this could also include informal textual descriptions.

Feature models Generative Programming puts emphasis on feature models as an important contribution to domain modeling and places feature models outside of the other concept models. Feature models describe the meaningful combinations of features and functions of the products in the domain, thus the

commonality and variability of the software family. Further details on feature models, which are essential in the research topic proposed in this report, are found in section 5.

Domain design is the subsequent activity of domain analysis that takes the domain model, and develops an architecture and production plan for the family of system to be built from the assets in the domain. The architectural design resulting from this activity prescribes how the components and assets are to be assembled to satisfy the requirements that can be posed on the family. The architecture has to be constructed in order to accommodate all the variability possible in the family. Since the architecture is a description of the components available in the system family and the composition constraints placed on their interactions, one can see a close connection to the descriptions in the feature model. The architecture should not only consider functional descriptions, but also non-functional requirements, such as performance, compatibility and so on.

The second artifact of the domain design activity is the production plan which describes the process of how to assemble the components, how to handle change requests, custom development and adoption of the assets to special requirements and the evolution of the measurements and processes used in the development of incarnations of the family.

Once the domain design has completed, it is followed by the domain implementation phase which involves implementing the components, languages, development processes and other assets designed. It also involves building the infrastructure used to realize the reuse of the assets in the domain model. That is, the interfaces and storages to find, access and assemble the components to instantiations of the product family.

The Application Engineering workflow

The application engineering work builds products and configurations of the software family using the reusable assets that result from the domain engineering phases. The application engineering workflow is intended to be carried out in parallel with the domain engineering activities, but while there is one instance of the domain engineering workflow for each product family, the application engineering workflow exists in several instances, one for each product to produced in the software family. Figure 12 on page 24, should thus not be interpreted as if the application engineering and domain engineering activities are running synchronously with each other, but rather illustrates the flow of assets, results and information between the phases involved.

The domain engineering workflow is iterative and constantly updates the assets and expands the capabilities of the organization with regard to what sort of products that the organization can provide. The process of developing applications based on the reusable assets is iterative as well, not only in that the product is released in new

and updated versions i.e. maintenance, but also in the application of iterative software development processes to the development of the applications. For instance, the application might be developed using prototyping, where new versions of the prototype system are produced as assets are implemented in the domain implementation phase.

The phases in application engineering correspond well to the ones that we find in most single application and software development methodologies [39]. The initial requirements analysis, takes the customer requirements and matches them to the set of capabilities that the software family can fulfill using the domain model. The requirements that can not be fulfilled using the resources in the domain model and domain assets are fed to the domain analysis phase in order to determine whether those requirements should be accommodated in the software family. If that is the case, the domain engineering activities creates the reusable assets to satisfy the requirements and store them to be used for future application engineering activities.

While the domain engineering approach to reuse of software intends to use the assets to the greatest extent possible, the software family assets can not possibly accommodate every configurability option. There will inevitably be concrete customer requirements that can not be fulfilled using the resources from the family domain engineering, and which will not be suitable for inclusion in the domain model or the reusable assets, being specific to the particular product. These requirements will require and trigger custom design and development specific for the current product.

The result of the application design phase is the software architecture that will be used to accommodate the reusable assets that fulfill the customer and system requirements. At the last leg of application engineering, the actual product is configured and instantiated using the reusable assets. The assembly of the assets can be manual, automated or use a semi-automated approach. Depending on how suitable the software components are and how evolved the organization is, the instantiation of the assets could be done using generators, code configurators or other advanced techniques.

4.1.3 Families

Financed by the European Union a European initiative was launched to do much of the same that was done in the STARS program. The projects ARES and PRAISE were launched in the mid-90s and followed by ESAPS [15] and CAFÉ [6], which in turn were followed by FAMILIES [16], which is ongoing and aiming to disseminate the results of the projects coherently. As in STARS, all projects have been carried out in cooperation between universities, research institutes and companies all over Europe. While the European and American research communities used to hold separate conferences, they eventually merged to coordinate the efforts better. The European projects are quite ambitious and take aim at providing a complete research framework for all aspects of product lines similar to the SEI Product Line Initiative. The

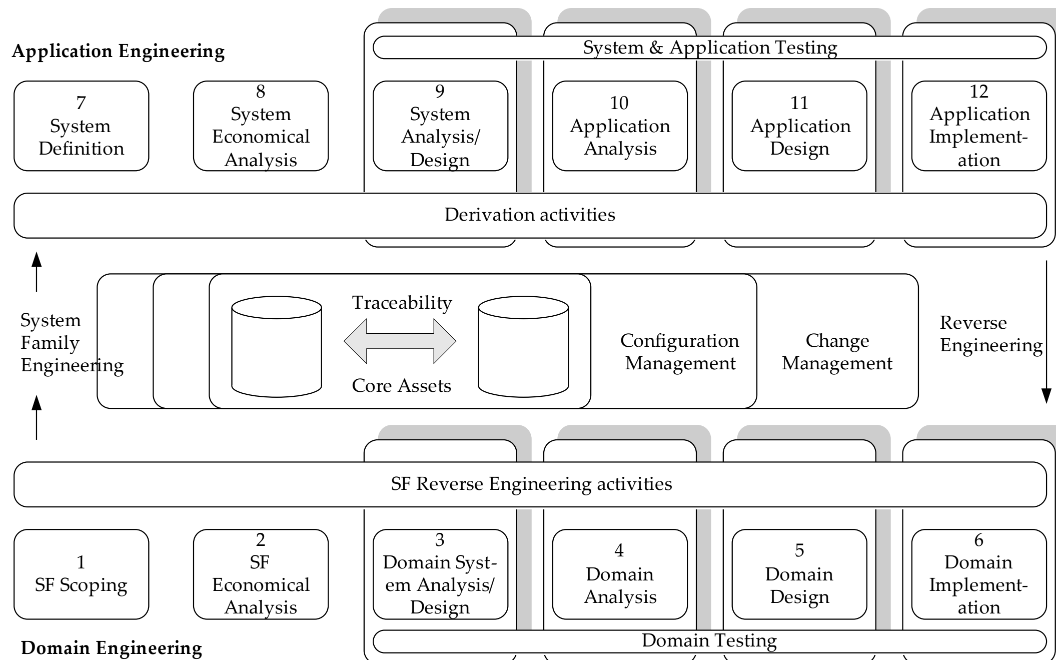

Figure 13: The System Family Engineering methods and processes. Adapted from [16].

work done in the European projects involve research on design and development of product lines, economic and quality factors, maturity models and more.

The European projects have resulted in a great amount of material concerning most aspects of the work concerning domain-based reuse. Obviously the material is similar to the SEI results, with various changes and modifications. Figure 13 shows the System Family Engineering (SFE) process, which resembles the process seen in Figure 12, although containing more details and elaborations. Since the academic and industrial partners have developed several different methods, there is no single process recommendation described yet, but rather many specific versions targeted at the needs of the different project partners.

In [3], activities that are part of domain analysis are mentioned as conceptual modeling, requirements modeling, commonality and variability analysis, domain modeling, feature modeling, scenario modeling and design mechanism capture. These are accompanied by the scoping activities domain scoping, product line scoping and asset scoping. These parts are incorporated to various extents in the different methods and work processes developed by the institutions and project partners that were part of the project. Most of the activities are dually related to the activities described in the previous sections, and so we will simply say that the generalization of the various specific processes fits to the generalizations made previously.

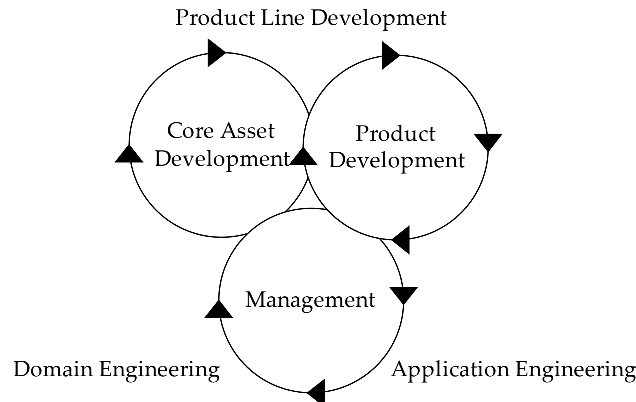

Figure 14: The overview of core activities in Software Product Lines. Adapted from [8].

4.2 Software Product Lines

Software Product Lines (SPL) originate in the domain analysis and software reuse projects of the early 1990s, but has taken a somewhat separate path than other projects and their descendants. Based on industrial experiences gathered by researchers at the Software Engineering Institute at Carnegie-Mellon University, SPL is a methodology that utilizes domain engineering and software reuse principles, and is still very active. The material about SPLs is very exhaustive as the knowledge and experience of researchers and practitioners has accumulated over the years. The handbooks and literature on the topic are exhaustive and detailed, and there are many publications on SPLs every year. The main outlet of information pertaining SPL is the website [49] and the documentation found there. Most of the information in this subsection is adapted from [8].

While SPLs still involve the same steps in one form or another as the methodologies that subscribe to the roots of domain analysis, SPL put a lot of effort into the aspect of management of the activities and the coordination and supervision of the phases involved in domain-based reuse. SPL-methodology also emphasizes the iterative nature of the execution of the activities, as well as the strong interactions between the activities. It is also understood that the activities do not necessarily follow a straight flow of events, but can occur in any order. This view is illustrated by the process diagram seen in Figure 14.

SPL basically consist of three activities that are run independently of, but interacting with, each other. The additional activity which SPL brings to the table, and which is only hinted at in Families, is the management aspect. Along with core asset development, management constitutes domain engineering and along with product development, it constitutes application engineering. Each of the three core activities

will be described below, bringing out the interesting points related to modeling and reuse aspects.

Core Asset Development

The core asset development activity results in a number of outputs, similar to those already mentioned about previous methodologies. The product line scope is the description of the products that the product line will result in, or that it will be able to include. The scope could be simply the enumeration of the names of the products that would be part of the product line, or it could be a more elaborate description of the products and what functionality, features and abilities they offer. The scope of the product line evolves and is modified based on changes in organization, market and other factors affecting the capabilities of the product line.

The second output of the core asset development is the core assets themselves. Among the assets are the architecture, software components, domain models, requirements specifications and assorted documentation belonging to the software. Each core asset has a process attached to it that specifies how the asset is to be used. Part of each core asset is also a plan for how the asset will evolve as the product line evolves. Although the core assets are commonly technical in nature, they can also include elements pertaining to risk identification, training and similar.

The third output of core asset development is the production plan, which is the description of how to accommodate the variation of the product line during production of applications from the core assets. The production plan describes the assembling, adoption and utilization of the core assets within the limits of the product line. The level of detail in the plan can vary from detailed process descriptions to informal guidelines.

The inputs to the core asset development-activity consists of product constraints, such as descriptions of commonality and variability, standards to adhere to, functional and behavioral features, physical constraints, quality considerations etc. In the same vein as product constraints are production constraints specifying infrastructures, legacies and production standards etc. which has to be taken under consideration when using the assets for producing applications.

The production strategy is the approach for producing the core assets, describing how the financing of the development of the core assets will be distributed, the general strategy to developing the asset base i.e. bottom-up or top-down. The latter decision would be based on another of the inputs, namely the inventory of preexisting assets. These assets would be harvested from the existing systems that the organization has built previously. While software elements, such as architectures, frameworks, libraries and similar would be the most obvious candidates for assets, much of the other intellectual property that an organization possesses, such as algorithms, training material and funding models, could be eligible. Lastly, SPL lists styles, patterns and frameworks as input to the core assets development. These are

inputs to be used for the design and construction of the architecture that forms the backbone of the software product line.

Product Development

The product development activity corresponds quite directly to what has been stated previously about the application engineering workflow. SPLs however, emphasize that the production of the applications will be guided by the production plan and will to some extent depend on the level of elaboration that the production plan possess. Should the plan be a detailed description, the product builders instantiate the production process and identify the commonality and variability of the product compared to what the product line offers. Should the production plan be more vaguely defined, the builders would have to devise a more detailed production plan that follows the advice and guidance given in the production plan. This is a slight step away from the more laid-out application strategy for the methodology described by Generative Programming.

There is also a difference in that SPLs dictates that there is more feedback going on between the application engineering and the core asset development than what is the case in Generative Programming, which does not have any feedback indicated from the application engineering workflow to the domain engineering workflow, apart from the feed of original customer requirements from the requirements analysis to the domain analysis step. While it makes sense that custom development of the application to fulfill the original customer requirements would result in software artifacts that could be used in the domain implementation of the requirements that were fed to domain analysis, there is no such step or information flow clearly indicated in the ”traditional” methodologies for domain engineering.

Management

One notable characteristic of the SPLs, is the emphasis placed on management of the SPL-process. Based on the experience of having introduced SPLs in several different organizations, the people behind SPLs have reached the conclusion that the use of domain-based software reuse is to a great extent a matter of managerial factors. The importance of having a strong leadership and persistence in times of difficulty during the introduction of domain-based reuse, is stressed throughout the material on SPLs. Although the entry level for SPLs might seem low with regard to the freedom of entry mentioned in the previous paragraph, it is considered imperative that the organization is wholly aimed at incorporating SPLs in all aspects of the software production. Key to accomplish this is the leadership and management of the SPLs. The management activity also produces some core assets in the form of reusable budgets and schedules.

The SPL-approach is a bit more relaxed when it comes to entry levels and work flows. While some methodologies for domain-based software development provide more guidelines or instructions for execution order and performing the activities in-

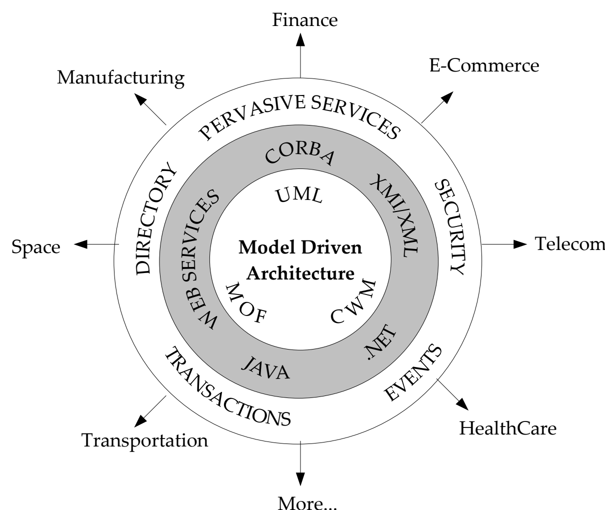

Figure 15: The MDA overview, describing the core and peels of the MDA. Adapted from [22].

volved in the methodology, SPL states that there is no given entry point to the essential product line activities, but an organization can begin at any point that fits to the organizations current status and maturity with respect to its ability to accommodate and draw benefit from the introduction of an SPL-based way of working.

4.3 Model Driven Architecture (MDA)

Taking the use of models in software engineering to the extreme, we find Model Driven Architecture [22]. MDA is the result of work conducted in Object Management Group (OMG) [23]. The goal of MDA is to increase portability, interoperability and separate logic from platforms. By extensive use of models, the architecture, design and implementations are modeled in platform-independent models, PIMs. These models are then targeted towards a platform and execution environment, and using tools for automatic translation, a platform-specific model (PSM) is generated. By creating business value in the models, they function as an insulating layer to changes in implementation technologies. MDA also includes other aspects, many aiming to overcome difficulties and interoperability problems on a technical level, concerning for instance communication between components and systems. Some of these concepts can be seen in Figure 15, along with some of the application domains for which there are standard domain models, providing the ability for rapid specialization.

The purpose is that most of the work that has to be done to develop an application should be contained in the PIMs and be reusable, in that they can effortlessly be re-targeted to another platform, using transformation rules. The models should be independent of programming languages and many other implementation issues,

such as development processes. By using advanced development tools, that integrate the modeling and the implementation management, the process should be further streamlined. In practice, the implementation material that is produced from the models, has to be completed with manually added material. The classical problem in this scenario is that as the model changes and the implementation is re-generated, the manual changes become inconsistent with the new implementation. The gains that are achievable are however quite large, provided that it is used in an appropriate project setting.

The modeling language of choice in MDA is the ubiquitous Unified Modeling Language (UML) [24]. UML is a standard that is published by the OMG, hence its tight connection to MDA. UML includes modeling constructs for practically every aspect of software development, including data flows, logical structures, behaviors and interactions and the constructs can be extended thanks to the refineable metamodel. UML has reached wide acceptance and use in industrial practice, and is supported by major vendors of software development tools. UML is useful as a modeling tool regardless of whether the organization employs the use of MDA or not.

MDA, along with UML and modern development environments, provide a very integrated work flow of modeling software. However, there is no clear consideration taken to the concept of software families thinking, other than that of products being available for different platforms. While it is thus possible to see transformations between different platforms as a form of variability, this is of course not the only variable in a product family. There has been some forays into developing methods for implementing variability and tie in product lines in MDA, see for instance [14].

5 Feature Models

In the previous sections of the report we have discussed the role of models in software engineering and requirements engineering. We have also discussed the premise of model-based software engineering and some approaches in MBSE. This section will present a particular type of model, which eminates from the approaches discussed in the previous section, and has capacity to be of use during the requirements engineering phases of development of software intensive systems.

The purpose of a feature model is to extract, structure and visualize the commonality and variability of a domain or set of products. Commonality is the properties of products that are shared among all the products in a set, placing the products in the same category or family. Variability is the elements of the products that differentiate and show the configuration options, choices and variation points that are possible between variants of the product, aimed to satisfy different customer needs and requirements. The variability and commonality is modeled as features and organized into a hierarchy of features and subfeatures, sometimes called feature tree, in the feature model. The hierarchy and other properties of the feature model are visualized in a feature diagram, see Figure 16 for a simple example.

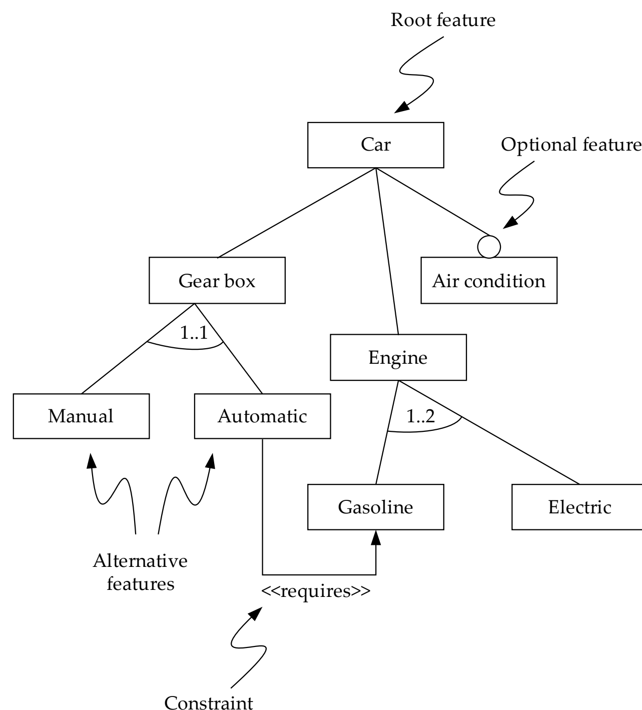

Figure 16: An example of a partial feature model describing a car.

5.1 History

The concept of feature models was originally introduced in 1990 by Kang et al with the FODA (Feature-Oriented Domain Analysis) technical report [31]. The original use for feature models was to facilitate domain analysis, but has since been used in other domains and for other purposes. The original definitions, notations and concepts used by FODA has been extended and modified as various other potential uses for feature models has become apparent. Many examples of requirement abstraction, architecture specification, etc. has been put forward over the years. [25, 35]

FODA was followed by the successor FORM (Feature-Oriented Reuse Method) [30]. A significant leap toward formalizing feature models was taken by Czarnecki and Eisenecker in Generative Programming [12], and further refinement of the feature model notation was published by Riebisch et al in [41] and [40]. Many others have made other contributions in various respects, and are mentioned in the remainder of this section where their contributions are discussed.

Using features as a way to describe software and system functionality is considerably older than FODA, although it was FODA that introduced structured modeling of features. The idea was incorporated in many of the techniques for domain modeling and domain analysis which appeared during the nineties, some of which briefly flashed into existence and went away, while others became successful and are used to this day. See section 4.1 for more on domain engineering. Today, feature modeling is an established and widely used technique which has been incorporated in many development methodologies and is the subject for interesting research.

5.2 Definitions

5.2.1 Features

The main concept of feature models is of course features. There are several definitions of the term feature used in conjunction with feature models. Some of them are more formal, while others are more intuitive. Features are intended to be concepts described by a single word or short line of text. These definitions are the ones most commonly used in literature:

- From FODA by Kang et al: ”A prominent or distinctive and user-visible aspect, quality, or characteristic of a software system or systems.” [31]
- From Generative Programming by Czarnecki and Eisenecker: ”Property of a domain concept, which is relevant to some domain stakeholder and is used to discriminate between concept instances.” [12]

- From IEEE: "A software characteristic specified or implied by requirements documentation (for example, functionality, performance, attributes, or design constraints). [26]
- From Riebisch et al: "A feature represents an aspect valuable to the customer." [40]
- From Bosch: "a logical unit of behavior that is specified by a set of functional and quality requirements." [5]

The distinction of user-visibility is interesting, as it place constraints on what to be considered features. One can argue that both the definition that considers user-visibility and the one that does not are equally valid, if viewed from different perspectives. From an applied, practical and user-oriented view it is of course meaningless to include features in a model that would not add to the users perception of the product [7,34]. The view of domain engineering would on the other hand want to include as much as possible of the domain information in a model, and would thus like to include all information relevant to any stakeholder. When discussing products that are part of larger systems in general and perhaps embedded systems in particular, the latter perspective seems to serve better. Although the FODA-definition lets the user be any stakeholder such as an actual person as well as a system, it only allows for direct influence, whereas the definition from Generative Programming allows more generic influences on stakeholders.

FODA listed three types of features:

Mandatory This type of features represent the common parts of a product, meaning that they are included in every configuration of a product where the parent of the feature is included. Mandatory features that are connected to the root concept and the mandatory features of those features, form a core or stem in the feature model, which represent functionality that is always included in all configurations of the product modeled.

Optional These features represent the variability of a product. Depending on the functionality needed in the configured product, a feature of this type may or may not be included, if the parent of the feature is included.

Alternative features Out of a set of alternative features, only one can be included in a product configuration, provided the parent of the alternative set is selected.

Each feature can either be a concrete feature that represent a concept that can be implemented and included in the product as a real function, or the feature could be an abstract feature. Abstract features, or pseudo-features, represent logical concepts that provide an abstraction or connection-point for a group of features, which in turn implements the abstraction and make it concrete.

Generative Programming [12] adds the type OR-features, which is similar to the alternative features type, except any non-empty set of the features can be included in the configuration, provided that the parent feature is included. This can then be combined with the other semantic types of features into a variety of different groupings like optional alternative features, optional or-features and optional alternative or-features, in order to allow for more variation in the minimum and maximum number of features selectable for inclusion in the configuration.

There is also the concept of parameterized features which at inclusion are assigned or assumes a value of some type. An example would be the sampling interval of a sensor, or the fuel consumption of an engine. Assigning a type/value-pair could of course be extended to several type/value-pairs, in which case it would start to resemble supplementary information described later in this report.

5.2.2 Feature Classification

There has been many different suggestions for classifications and categories of features. The original FODA discuss features categorized accordingly:

- Operating environments in which applications are used and operated.
- Capabilities of applications from an end-users perspective.
- Domain technology common to applications in a particular domain, exemplified by navigation algorithms from avionics.
- Implementation techniques based on design decisions.

No further motivation is given for this choice of feature classifications, other than that they make sense in many common cases. Based on taxonomies from user interface design, the capabilities class is further refined into functional features, operational features, and presentation features [31].

Yet another set of categories, referenced in [12], proposed by Bailin talks of operational, interface, functional, performance, development methodology, design and implementation features. Riebisch et al suggests a distinction of functional features expressing the way users interact with the product, interface features expressing the conformance to a standard or sub-system, and parameter features expressing enumerable, listable environmental or non-functional properties. The grounds of this classification is that the feature model should serve as a customer view and that all other information should be captured in other models during design and implementation. The proposed classification therefore reflects the customers view on a product and in what terms a customer considers a products functions and capabilities.

Whereas one could argue about what classification that is the most useful, it seems quite apparent that it is the domain of investigation and the application of the modeling technique that should guide which classification that is most suitable. There is no reason to use a classification scheme that does not make sense in the current context, and one could thus imagine many other ways of categorizing features in order to assist in the production and use of the feature model. All methodologies for feature and domain analysis recognize this very fact, and suggest that the categorization should be done based on the experiences in the domain and according to the intended use and context of the feature model.

Interestingly, no author makes a clear-cut argument for the usefulness and purpose of classifying features, or make any statements about the benefits that would come from using a classification scheme. There seems to be a general agreement among the authors mentioned above that the classification should serve as some sort of view on the system from the perspective of some stakeholder. This idea on classification does not seem very well thought trough though, since the classifications often does not come across as particularly striking for any well-defined set of stakeholders. In other words, the classification schemes seem more like a relic from a requirements engineering work pattern.

5.2.3 Constraints and Relations

The organization of the features into a hierarchy often seem intuitive at first glance, but at closer examination the semantics of the hierarchical structure can be a bit confusing. The general idea of the hierarchy is to structure the features in such a way that moving down the tree, higher degrees of detail is achieved and more detailed decisions on design are made. FODA hints that the semantics of the hierarchy is ”consist-of”, and in FORM there are three different kinds of semantics, namely ”composed-of”, ”generalization/specialization” and ”implemented-by”.

Generative Programming states that the semantics of the hierarchy can not be interpreted unless considering the features related and their types. The lack of semantics in the feature model is deliberate and the authors instead say that structural semantics should be placed in another model, more suitable for it such as ER-models.

Apart from the relations that arise as a consequence of the use of mandatory and optional features in a feature hierarchy, features in a model can also have other interdependencies to one another. FODA originally described two types of feature interdependencies, namely ”requires” and ”mutually-exclusive-with”. These are hard constraints used to, for instance, indicate that a manual transmission in an automobile is mutually exclusive with an automatic transmission. Whereas these constraints are sufficient for most purposes, they do not offer much flexibility and there could be modeling constructs that can not be accommodated using these basic relation stereotypes [40].

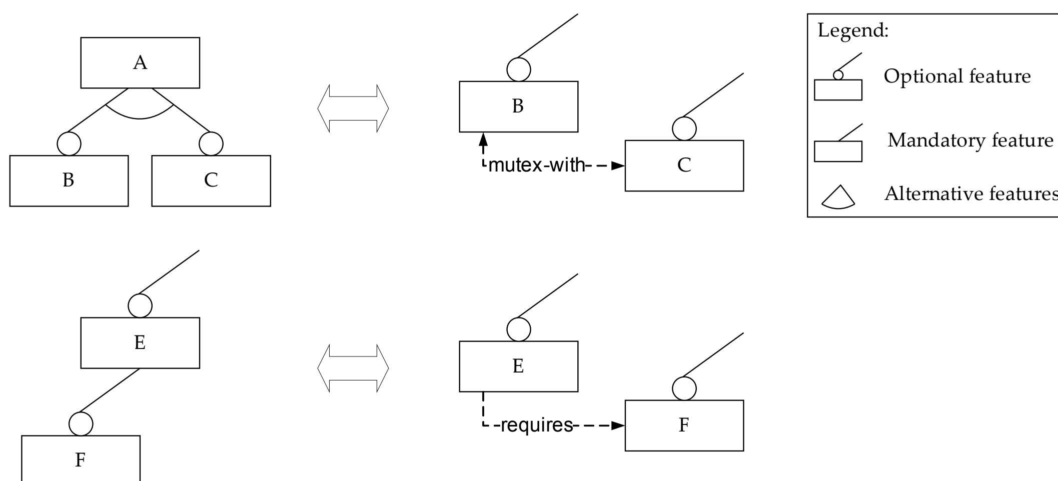

Figure 17: The equivalent constructions of the mutex and requires relations. Adapted from [40].

In [18], the authors construe the ”provided-by” relation used to realize an abstract feature into a regular implementationable feature. They also rename the ”consists-of” relation to ”refine”, and also loosen the boundaries between hard and soft constraints. Riebisch notes that the semantic difference between the relation that make up the hierarchy and the ”require” relation is quite small. Also, the ”mutex-with” and the alternative relations are similar in semantics and constitute different ways of achieving similar constructions in the feature model. See Figure 17 for an example of a transition. (The specifics of the notation is explained later in the next subsection.) Riebisch groups ”is-a”/”part-of” along with ”requires” into the hierarchy, based on the features that exist at the endpoints of the relation.

Generative Programming introduces so called weak constraints, which can be used to indicate default dependencies for features, but can be overridden if necessary. Riebisch calls this particular type of relation for hint relation. Riebisch also adds the refinement stereotype, which is used to point to features that are significantly effected by the inclusion of a certain feature. Several other authors has added other kinds of constraints, relations and tracing mechanisms to feature models. [12, 17, 40, 43]

In theory, one could adorn feature models with any amount of information concerning dependencies and tracing between features, but in practice the models tend to expand quite rapidly for anything but the simplest systems, even using the basic stereotypes. Using stereotypes is however a powerful means of expanding the semantics and abilities of the feature model. It is not unreasonable to believe that there could be particular domains where the addition of a special semantic construct would be meaningful and essential for the understanding and usefulness of the feature model.

While the original semantics of feature models was not very well-defined to begin with, further extensions and modifications of the feature models has resulted in more and more semantic meanings being imposed on the relations, without really clarifying much. As the original tree structure is pushed back in favor of more general directed graphs with equivalent types of edges, the matter of clearing up the semantics is an interesting research question.

5.3 Feature Diagram Notations

The feature diagram is an essential part of a feature model. The feature diagram is the visualization of the feature model. It is a hierarchical decomposition of the features in the model, indicating dependencies and constraints for the commonality and variability of the product that is represented by the feature model. It usually has the form of a tree structure with the root node of the tree representing the concept that is described by the rest of the tree. The nodes and edges of the tree are usually decorated in a particular notation in order to indicate the dependencies and constraints placed on the features. It is not necessary to have the feature model organized into a tree structure, as there is nothing in the semantics of the relations that can be made between the features that restricts the constructions. It is possible to let the model take on a more general graph structure, but a tree is usually seen as more useful as it indicates more clearly the distinction between the different partitions of features in the model, as well as the further levels of detail that is added as the user of the model makes the choices of features.

The information that is visualized in the feature diagram could easily and conveniently be contained in some other format for processing and storage, however one of the very points of models is the visualization aspect. We will therefore use the feature diagrams to introduce some of the semantics that has been introduced in feature models by researchers.

There are substantial amounts of information available concerning the notation of feature diagrams, as well as information suitable to be supplemented in a feature model. For a more detailed comparison of notations, see [48]. The usefulness and necessity of the extensions made to the feature diagrams is debatable. While in some cases it certainly adds clarity and brevity to the notation, it is argued in [4] that they do not add any more expressiveness to the diagrams. This of course is not the case if the extension adds notations and semantics that is brand new and did not exist in the first place, such as the hint stereotype.

5.3.1 FODA

The original notation for feature diagrams comes from [31] and contains the basic building blocks of feature models, such as mandatory, optional and alternative

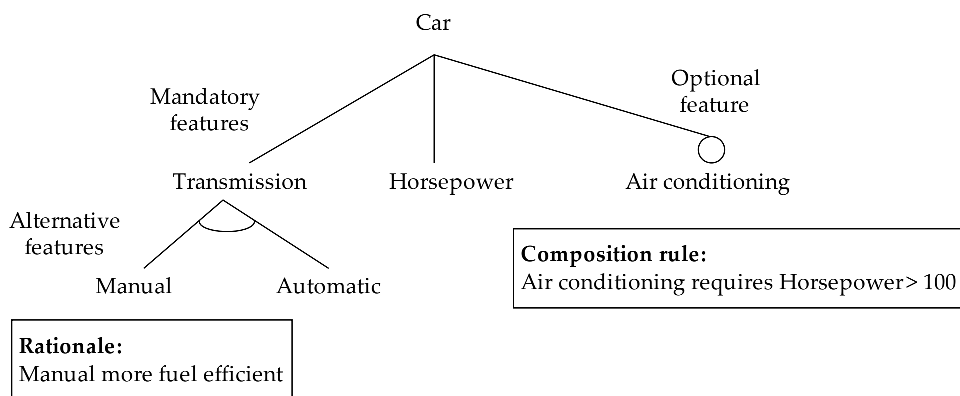

Figure 18: The FODA-notation of feature diagrams. Adapted from [31].

features and some composition rules such as dependency and mutual exclusivity. Figure 18 illustrate the notation for mandatory, optional and alternative features, as well as some composition rules. As the constructs in this first version of feature models are quite simple and fundamental, the notation and appearance of the feature diagrams is also un-elaborative and easy to understand intuitively.

Figure 18 also illustrates how supplementary information about composition rules and rationale is included as textual information alongside the feature diagram, rather than as edges in the diagram with stereotypes added, which we will see in later revisions of feature diagrams. Optional features are in this notation denoted by an empty circle, and groups of alternative features are denoted by an empty arc connecting the edges to the groups features.

The changes made in the successor of FODA, called FORM, introduced four layers in the feature model denoted capability layer, operating environment layer, domain technology layer and implementation technique layer. As seen in Figure 19, FORM uses the interdependencies types generalization/specialization, composition and implementation, tracing these interdependencies across layers in the model. The resulting diagrams very quickly become hard to overview, but the problem is to some extent mitigated by not including the composition rules in the diagram. The decorations of the features are the same as those in FODA. FORM represents a substantial expansion of FODA, not so much in notation as in the application of the feature model, and the target of the model. By layering the information, the features are grouped in a way that lets the user call decisions on design and implementation issues. While these issues can be contained in features regardless of notation, it is more notably expressed in the FORM-notation.

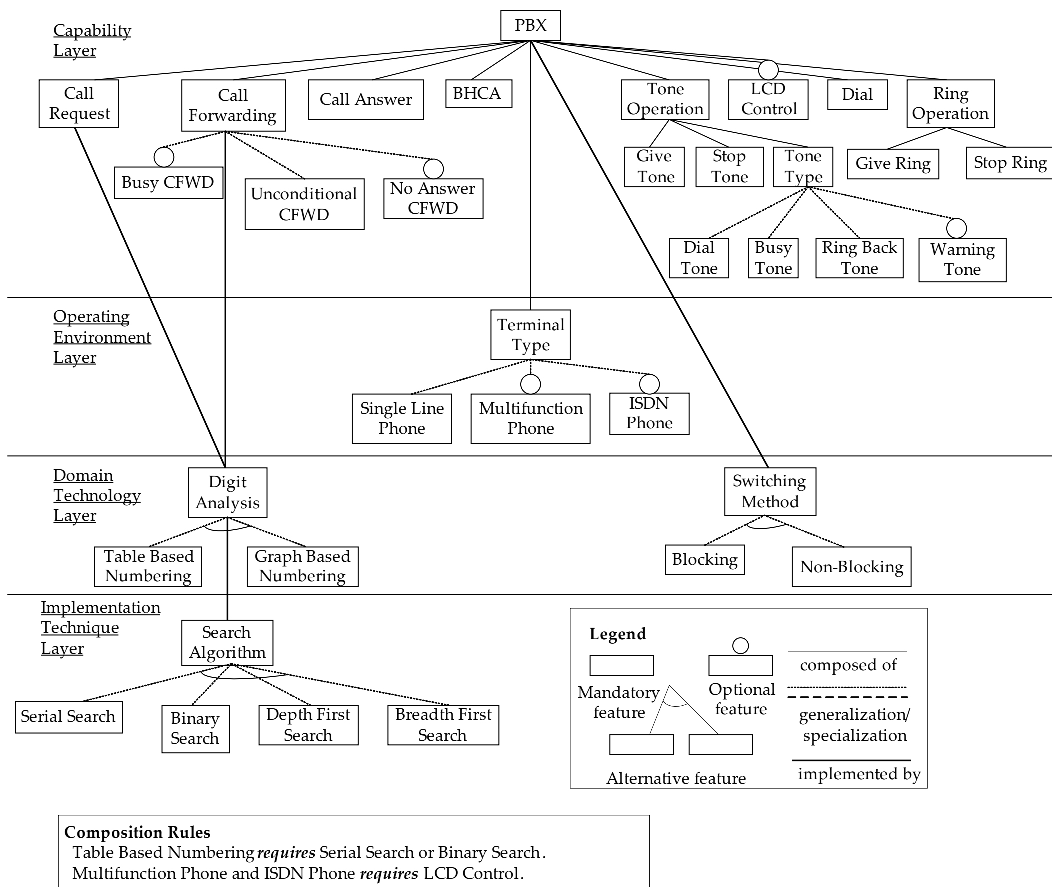

Figure 19: The FORM-notation of feature diagrams. Adapted from [34].

5.3.2 Czarnecki and Eisenecker

In Generative Programming by Czarnecki and Eisenecker [12], the FODA-notation of feature diagrams is slightly modified and also extended to include OR-features. OR-features are features that are non-exclusive to each other, and one can thus include several of the features denoted to be OR-features rather than merely including one of the alternative features. Figure 20 illustrates the notation used by Czarnecki and Eisenecker. Mandatory features are here decorated with a filled circle, while optional features have an empty circle. OR-features are indicated using a filled arc between the edges of the group of alternative features.

5.3.3 Riebisch

In [40], Riebisch makes further extensions to feature models by adding more stereotypes to relations between features such as hints and refinement. He also suggested

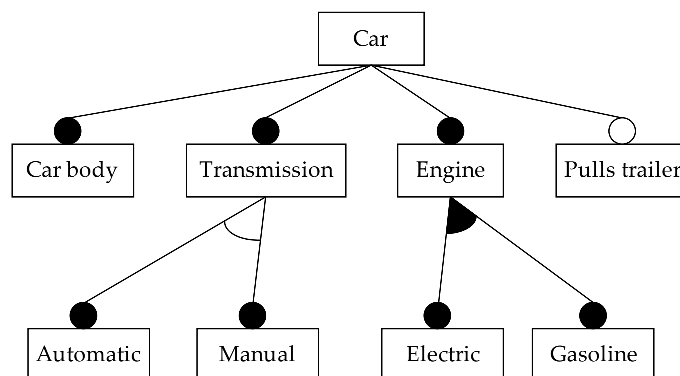

Figure 20: The Eisenecker-notation of feature diagrams. Adapted from [11].

changes in [41], with the use of multiplicities to denote the choices of features. Figure 21 illustrates some of the changes and additions made by Riebisch.

The previous notations used the composition rule of feature/subfeature and alternative features to implicitly indicate the relations requires and excludes, and complemented this with the use of textual information describing composition rules. In Riebisch notation, stereotypes are added to additional edges between features to indicate these types of dependencies. This makes it possible to model the features a bit more freely in the decomposition hierarchy, while not sacrificing the possibility to denote hard and soft constraints in the diagram. The choice of whether to include such constraints as objects in the diagram or whether to use information in textual or other formats apart from the diagram, is up to the user and the tools that are available to facilitate, not only the construction of the model, but also the use and understandability of it.

A notation for parameterized features is also introduced in this version of the feature diagram notation. By using multiplicities in the arcs of optional groups, the notation is cleaned up, and the need for OR-features and alternative feature groups is eliminated. The use of multiplicities is similar to that in for instance UML, where a range on the form 2..* indicates that at least two features has to be selected, and the maximum number of features allowed to be selected is unbound up to the total number of features in the group. Although Riebisch argues that this notation removes ambiguities that were present in previous notations concerning optional feature groups, other disagree and claim that the notation does not add any more expressiveness to the semantics compared to previous versions. In practice, the notation is perceived as more easily understood, and there is an obvious familiarity aspect as the notation reminisce to UML and ER-modeling.

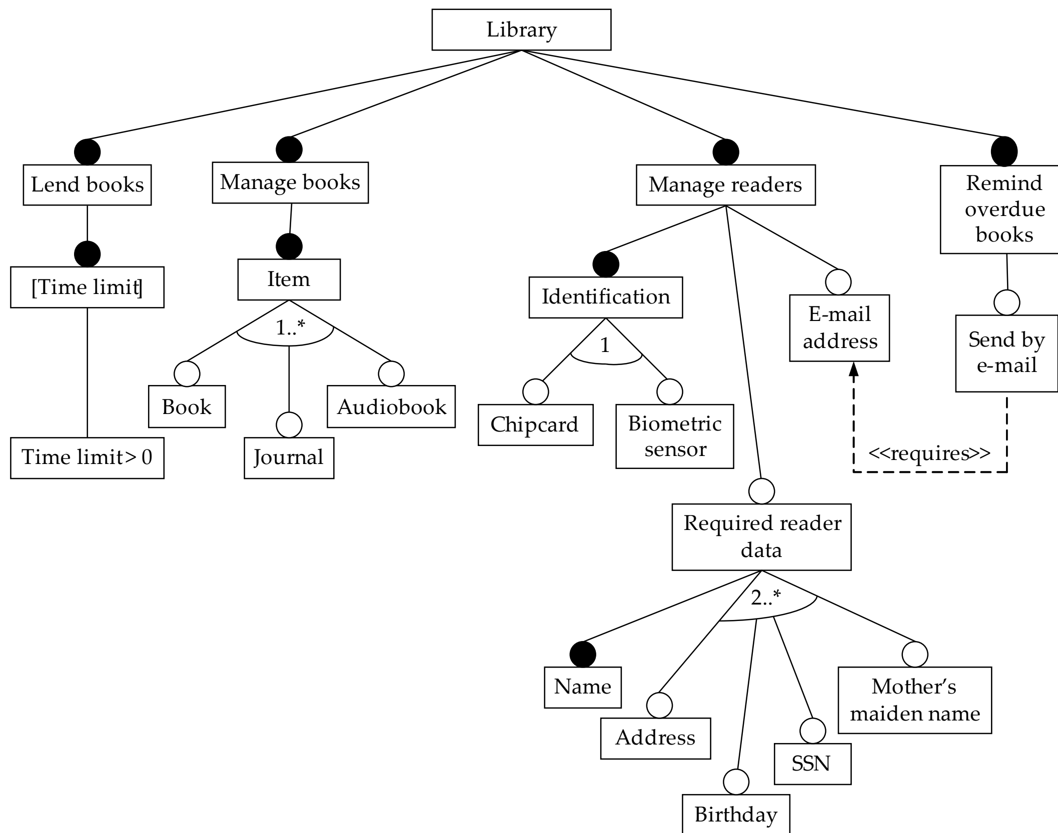

Figure 21: The Riebisch-notation of feature diagrams. Adapted from [41].

5.3.4 Supplemental Information and Attributes

The additional composition rules in a feature model, apart from the ones that are implicit from the use of the hierarchy in the diagram, can as mentioned previously be kept in textual form in supplementary material to the diagram. It helps keeping the diagram cleaner and less cluttered, while making it easier to present the material in a more accessible format.

Apart from the composition rules mentioned by FODA and FORM, several additions has been made as researchers has applied the modeling technique in industrial settings. The relevance of some of the entries suggested to be part of the supplementary information might depend on the purpose and context that modeling takes place in.

Czarnecki also elaborates considerably on the use of supplementary information for each feature in the model, among others semantic description, rationale explaining why the feature is included and when the feature should be selected, stakeholders that has an interest in the feature, exemplar systems, constraints and priority. The volume of information makes it unsuited to be contained in the feature diagram and is

instead maintained in a separate document accompanying the feature diagram. These attributes provide means for storing information in the model in order to widen the field of application for the feature models. While the purpose of modeling commonality and variability is still the focus of the model, additional information makes it possible for more stakeholders to use the model in more contexts, where variability of the product is a useful abstraction.

We list some of the more common and anticipatable useful entries here, mainly as mentioned in [12].

Semantic description Each feature should have a short description of its semantics.

Rationale An explanation of why the feature is included in the model and annotation of when the feature should be selected and when it should not be selected.

Stakeholders Each feature should have an annotation of which users, developers, software components, customers, client programs etc., that has an interest in the feature.

Exemplar systems If possible, a feature could have a note on existing systems that implements this feature.

Constraints Constraints are hard dependencies between variable features that dictate which features that are necessary to include in order to ensure functionality. There can also be recommendations or hints on features to be included in a description, given the inclusion of a particular feature.

Availability and binding sites/mode An availability site describes when, where and to whom a feature is available. An available variable feature has to be bound before it can be used. A binding site is the site where a variable feature may be bound. A site model could consist of predefined times, user actions, contexts, etc. The binding mode describes if the binding of the feature is static, changeable or dynamic. Binding site can serve as yet another way of classifying features in a model.

Priority A feature can have a priority assigned to it in order to reflect its relevance to the project or software line.

There are several other entries that has been suggested. One is to allow a feature to be associated to a particular type such as integer or string which would allow the feature to assume a value. This is also referred to as attributes and allows a type/value-pair to be associated with each feature as mentioned previously. Fey et al. [18] talks of properties that are associated with features and that let the feature take on values in the same way. They go one step further though and discuss the possibility of dependencies and interactions between properties as well as between

features. The interactions of the feature properties is described as some form of information flow between the features and the setting of one property in feature to a particular value would affect another feature through a ”modify” relation. To some extent the feature diagram is at this stage turning into something of a design model.

Some of the entries in the supplementary information are visible in the feature diagram, such as the constraints information and dependencies. The reason to include it also in written or non-pictorial form is to make it possible to add complex interdependencies that does not lend it self to the graphical format. There can also be benefits of having the same information written in a formal notation in textual form in order to allow analysis of the feature model by tools.

5.4 Methodologies and Uses of Feature Models

The original use for feature models as a means and aid to perform domain analysis was over time complemented with other uses, as the structuring of properties of a domain or product set into features turned out to be an efficient and communicative representation. Apart from domain analysis, there are two principal categories of uses, which most effort can be sorted under.

The first is to support the requirements management process for lines or sets of products. By letting the capabilities of a product line be represented by features, requirements posed on the product by customers, as well as internal requirements, can be abstracted into features. By letting a set of common requirements on a product be represented by a feature, one can achieve higher degrees of reuse. Using this approach, feature models can be used as a means of communicating information about the product to customers during requirements negotiation or sales, conveying implementation requirements to developers or a means to exchange information between other stakeholders.

The second potential for feature models is to use them for configuration and automated construction of an instantiation from the product line described by the model. Ideally, this would mean that each feature would principally represent one or more components or source-level packages used to add functionality to the instance. The selection of features from the feature model would thus guide automated scripts, which would build the product with the requested functionality. [50]

There are several methods and guidelines for domain analysis that use feature models in one way or another. This section discusses some of these ways and provides a summary of the steps taken to perform the analysis process resulting in a feature model. Generally, they are the same steps as in domain engineering or requirements engineering methodologies and they follow the same principal pattern. Apart from the methods treated here, more information is available in for instance [9, 42].

5.4.1 FODA/FORM

FODA and its successor FORM uses features to perform domain analysis and share most characteristics with each other. Most subsequent methodologies for feature analysis are heavily based on FODA, which makes understanding of the FODA-process a prerequisite to understand most methodologies on the subject. The general feature analysis process consists of (1) collecting information sources, (2) identifying features, (3) abstracting and classifying features into a model, (4) defining the features and (5) validating the model.

Among the information sources used for finding features is documentation, such as user manuals, requirements documents, design specifications, implementation documentation and source code. Apart from product documents, one can also use standards, textbook material and also domain experts. When processing the sources for potential features one should take care to resolve ambiguity in meaning of concepts. Understanding the language of a domain is generally regarded as an effective way of finding possibilities for features.

Once features are identified, they should be classified and structured into a hierarchical model using the consists-of relationship. During the modeling, each feature should be indicated as being mandatory, optional, alternative, etc. Each feature should also have resolved dependencies, and should be supplied with additional information. In order to ensure that the feature model is made as complete and useful as possible, it should contain features from both high levels of abstraction such as functional and operational features, as well as more technical features representing implementation details. The structure of the feature hierarchy could be quite varied. For instance, one should consider whether two features in different parts of the feature model, which are mutually exclusive should instead be organized as neighboring alternative features.

Once the model has been completed, it should be validated against existing applications and by domain experts. Preferably the validation should be made by domain experts that were not part of the construction of the model, since they are less likely to be biased. It could also be useful to validate the model against at least one application that was not part of the material analyzed for the model.

Further details on the methodology is found in [30, 31].

5.4.2 The Generative Programming Feature Finding Process

Czarnecki and Eisenecker provides a feature modeling process in [12], which identifies sources for features, discusses strategies for finding features, and the general steps taken in feature modeling. This section describes the main points of the material covered in the book.

As sources of features, the authors mention not only existing, but also potential stakeholders, domain information available from domain literature and domain experts, existing systems, and existing models. The existing models could be available object models, use-case models and other models created during design and implementation phases of the software development. For identifying features, it is important to remember that anything that a user might want to control about a concept could be a feature. Czarnecki and Eisenecker therefore consider implementation techniques and other implementation issues as features to be considered.

The book suggests looking for features at all points in development and investigate more features than what are intended to be initially implemented in order to give room for some growth in the future. At some point in the process there should be a scoping activity where the features to be implemented are settled on. At this point the use of priorities for the features are important.

The book describes a ”micro-cycle” of feature modeling, which follows the standard workflow of identifying similarities between all instances of the products, record the differences between instances, that is variable features, and then organize them into diagrams. After this the feature interaction analysis ensues, during which contradicting dependencies are resolved. This could lead to the discovery of feature interactions and combinations that were not discovered initially. This workflow is similar to that of requirements engineering.

5.4.3 PuLSE-CaVE

PuLSE-CaVE [29] is an approach developed to integrate textual information contained in documentation for legacy systems into a domain model for product lines. Although its focus is to extract information from documentation and primarily from user manuals, it aims to elucidate information concerning commonality and variability, which is of course the main objective of creating feature models, making the methodology a bit interesting to us. It also explicitly treats the issue of extracting requirements from documentation.

The process employed by PuLSE-CaVE to elucidate information is divided into three phases, preparation, search, and selection, change and modification. The preparation consists of collecting, selecting, dividing and browsing the documentation in order to structure the documents into manageable work packages. PuLSE-CaVE lists some guidelines on how to reasonably identify features from textual information, as well as concrete hints based on heuristics that can be used to guide the analysis of the documentation. These hints are generally of the character ”phrases that differ in only one or a few words can be evidence for alternatives”.

The last phase is selection and validation of the extracted information, in order to structure them into a model. Since the methodology is not focused exclusively on identifying feature elements in the source information, the selection of found

elements is guided by what model one is using, and what type of elements that model uses as primitives.

6 Conclusions and Research Questions

Although software engineering has had quite some time to mature and create solid foundations and tools for use in the engineering work, there are still lots of room for improvement. As stated in the beginning of the report, the requirements engineering process, models and domain understanding are the basis for the work that originates in this report.

Requirements engineering repeatedly proves itself to be a complex and critical activity in the development of software-intensive systems. Improvements to this activity can be of significant value, and understanding of the problems and the needs for effective requirements engineering is of great interest to reach those improvements. Research done about domain modeling and facilities for domain understanding has taken various directions with various results. While Software Product Lines has taken prominence as a suitable and applicable methodology for successful development of application families, there is still much to be done to bring the research of software engineering into a practical form.

This research will attempt to take a part of the work done in model-based software engineering and bring it into a practical form, that can be applied in a selected context. The premise is to use feature models to structure the functions and capabilities of the products and product derivatives in an organization. The discursion will then be how these models can be integrated in the requirements engineering process, in order to support modeling of requirements in a manner that can propagate the benefits of modeling throughout the development process. Examples of effects that would be interesting to investigate are if the process will result in higher quality of the requirements, faster progression through the process, better understanding of the requirements and less ambiguity in the requirements specifications. Other interesting dimensions are the facilitation of requirements reuse and model use for change request management.

The following research questions encompass all the interesting elements of the research brought up in this report.

What is a suitable meta-model for describing the commonality and variability for requirements engineering?

How does one develop such a model efficiently?

How does one utilize such a model efficiently?

References

[1] Technical Report STARS-AC-04110/001/00, Paramax Systems Corporation, 1992.

[2] Scott W. Ambler. *Agile Modeling*. Wiley Computer Publishing, 2002.

[3] Pierre America, Steffen Thiel, Stefan Ferber, and Martin Mergel. Introduction to domain analysis. Technical report, ESAPS, 2001.

[4] Y. Bontemps, P. Heymans, P.Y. Schobbens, and J.C. Trigaux. Semantics of FODA feature diagrams. In *The Third Software Product Line Conference (SPLC04)*, 2004.

[5] Jan Bosch. *Design and Use of Software Architectures*. Addison Wesley, 2000.

[6] CAFÉ. From Concepts to Application in System-Family Engineering. http://www.esi.es/Cafe/, March 2006.

[7] Gary Chastek, Patrick Donohoe, Kyo Chul Kang, and Steffen Thiel. Product line analysis: A practical introduction. Technical Report CMU/SEI-2001-TR-001, Carnegie-Mellon University, 2001.

[8] Paul Clements and Linda Northrop. *Software product lines: Practices and patterns*. Addison-Wesley, 2002.

[9] Dick Creps, Will Tracz, Teri Payton, and Mike Webb. Domain engineering handbook. Technical report, Lockheed Martin Software & Systems Resource Center, 1996.

[10] Richard E. Creps, Mark A. Simos, and Rubén Prieto-Díaz. The STARS conceptual framework for reuse processes. Technical report, 1992.

[11] K. Czarnecki and U. Eisenecker. Synthesizing objects. In *Proceedings of ECOOP'99*, 1999.

[12] K. Czarnecki and U.W. Eisenecker. *Generative Programming*. Addison Wesley, 2000.

[13] Krzysztof Czarnecki. Domain engineering. Technical Report DOI: 10.1002/0471028959.sof095, Encyclopedia of Software Engineering, 2002.

[14] Sybren Deelstra, Marco Sinnema, Jilles van Gurp, and Jan Bosch. Model driven architecture as approach to manage variability in software product families. In *MDAFA 2003*, 2003.

[15] ESAPS. Engineering Software Architectures, Processes and Platforms for System-Families. http://www.esi.es/esaps/, March 2006.

[16] ESI FAMILIES. Families. Main. http://www.esi.es/Families, March 2006.

[17] Stefan Ferber, Jürgen Haag, and Juha Savolainen. Feature interaction and dependencies: Modeling features for reengineering a legacy product line. In *SPLC2*, 2002.

[18] Dániel Fey, Róbert Fajta, and András Boros. Feature modeling: A meta-model to enhance usability and usefulness. In *SPLC2*, 2002.

[19] Robert France and Bernhard Rumpe. Assessing model quality. *Journal on Software and System Modeling*, 2004.

[20] Robert France and Bernhard Rumpe. Domain specific modeling. *Journal on Software and System Modeling*, 2005.

[21] Martin L. Griss, John Favaro, and Massimo d'Alessandro. Integrated feature modeling with the RSEB. In *International Conference on Software Reuse*, 1998.

[22] Object Management Group. MDA. http://www.omg.org/mda/, March 2006.

[23] Object Management Group. Object Management Group. http://www.omg.org, March 2006.

[24] Object Management Group. Object Management Group - UML. http://www.uml.org, March 2006.

[25] Andreas Hein, John MacGregor, and Steffen Thiel. Configuring software product line features. In *ECOOP 2001 Workshop on Feature Interaction in Composed Systems*, 2001.

[26] IEEE Standards Board. IEEE Standard Glossary of Software Engineering Terminology. Technical Report lEEE Std 610.121990, IEEE, 1990.

[27] Michael Jackson. The world and the machine. In *ICSE'95*, 1995.

[28] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. ACM Press, 1997.

[29] Isabel John and Jörg Dörr. Building domain models based on legacy system descriptions. Technical report, The Café project, 2003.

[30] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168, 1998.

[31] K.Kang, S.G.Cohen, J.A.Hess, W.E.Novak, and S.A.Peterson. Feature-oriented domain analysis (FODA) - feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon University, 1990.

[32] Gerald Kotonya and Ian Sommerville. *Requirements Engineering, Processes and Techniques*. Wiley, 1997.

[33] Ola Larses. *Architecting and Modeling Automotive Embedded Systems*. PhD thesis, Royal Institute of Technology, 2005.

[34] Kwanwoo Lee, Kyo C. Kang, and Jaejoon Lee. Concepts and guidelines of feature modeling for product line software engineering. In *The Seventh Reuse Conference*, 2002.

[35] Dongyun Liu and Hong Mei. Mapping requirements to software architecture by feature-orientation. In *STRAW03*, 2003.

[36] Jochen Ludewig. Models in software engineering - an introduction. *Journal on Software and System Modeling*, 2003.

[37] James Neighbors. *Software Construction Using Components*. PhD thesis, University of California, 1980.

[38] David Lorge Parnas. On the design and development of program families. *IEEE Trans. Software Eng.*, 2(1):1–9, 1976.

[39] Shari Lawrence Pfleeger. *Software Engineering: theory and practice*. Prentice-Hall, second edition, 2001.

[40] Matthias Riebisch. Towards a more precise definition of feature models. In M. Riebisch, J. O. Coplien, and D. Streitferdt, editors, *Modelling Variability for Object-Oriented Product Lines*. Norderstedt, 2003.

[41] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. Extending feature diagrams with UML multiplicities. In *6th Conference on Integrated Design & Process Technology*, 2002.

[42] Mark Simos, Dick Creps, Carol Klingler, Larry Levine, and Dean Allemang. Software technology for adaptable reliable systems (STARS) organization domain modeling (ODM) guidebook version 2.0. Technical Report STARS-VC-A025/001/00, Lockheed Martin Tactical Defense Systems, 1996.

[43] Periklis Sochos, Ilka Philippow, and Matthias Riebisch. Feature-oriented development of software product lines: Mapping feature models to the architecture. In *Object-Oriented and Internet-Based Technologies*, 2004.

[44] Ian Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 2001.

[45] Ian Sommerville and Pete Sawyer. *Requirements Engineering, A Good Practice Guide*. Wiley, 1997.

[46] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer Verlag, 1973.

[47] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press, second edition, 2002.

[48] J.C. Trigaux and P. Heymans. Modelling variability requirements in software product lines: A comparative survey. Technical report, Institut d'Informatique FUNDP, 2003.

[49] Carnegie Mellon University. Software Product Lines. http://www.sei.cmu.edu/productlines/, March 2006.

[50] Arie van Deursen, Merijn de Jonge, and Tobias Kuipers. Feature-based product line instantiation using source-level packages. In *SPLC2*, 2002.