Master's Thesis

Tool Assisted Product Configuration in Software Product Lines

Jacob Kristhammar D04 & Roger Schildmeijer D04

Department of Computer Science
Faculty of Engineering LTH
Lund University, 2008

ISSN 1650-2884
LU-CS-EX: 2008-20

Tool Assisted Product Configuration in Software Product Lines

Jacob Kristhammar (kristhammar@gmail.com)
Roger Schildmeijer (d04rp@student.lth.se)

August 8, 2008

Abstract

Software product lines enable large organizations to derive different products on a tight schedule by reusing its developed assets.

This master's thesis will identify some of the problems involved with product configuration in software product lines by investigating how it is done at Sony Ericsson Mobile Communications. We will use the identified problems to present how a tool could be used to improve the process of product configuration.

In order to fully benefit from the reuse of developed assets the variability mechanisms and asset structure must be well defined. Otherwise the derivation of products from a software product line can get very complex and difficult to manage. The number of variation points and dependencies among them give rise to problems that needs to be taken care off. Processes and responsibilities are other areas that are important in order to manage a software product line. Another issue with product configuration is to enable some level of traceability between the configured products and the system requirements to ensure that the right products are produced.

We have identified that products are configured in a bottom-up manner, i.e. the binding of the variation points are performed by the developers rather than a dedicated unit that is responsible for the derivation of products.

We present how product configuration could be done by using variability suggested by the requirement engineers and product planners. A product configurator prototype has been developed to demonstrate how the benefits from well defined variation points and their dependencies can be used to derive new products from the software product line and maintain existing ones. We also propose how the current configuration process should be changed and give examples of how some tasks could be done using the developed prototype.

Keywords

Product Configuration, Software Product Line, Product Family, Product derivation, Variability, Variation point, Tool, Configuration Management, Reusability, Traceability

Contents

Chapter 1

Introduction

In this chapter we will provide a general background about product configuration and the problems involved. A list of questions the master's thesis aim to solve and the methods used will be presented.

1.1 Background

The mobile phone market is evolving fast and the requirements customers put on the phones is changing from day to day. The demand for new phones is ever-increasing and it is crucial that the mobile manufacturers release new products at the same pace to not loose their market share. In addition to the demands on frequent releases it is important that the manufacturers offer a wide variety of phones to attract new customers and keep the ones they've got.

The development process needs to handle the great variety of phones and frequent releases. It would be infeasible to start developing from scratch each time a new phone is developed. Reinventing the wheel every time would be a painful and costly process. It is also important to minimize the time-to-market when developing a new product to avoid that the new functionality gets obsolete before the phone reaches the market.

To achieve this, many companies try to adopt techniques that let them develop their assets in parallel and enable a high degree of reuse. In a majority of software development projects, the ability to reuse core assets is important[4]. One way to achieve this is to group similar functionality into assets that different development units are responsible for. Different variants of products can later be derived by combining the assets. This is common in many organizations and is often referred to as a *software product line* or *software product family*. Different mechanisms to derive products from such product families exist, but it is a challenging task[16]. In order to utilize the benefits, like parallel development and reusability, the interfaces between the different assets an their relationships need to be well defined. It is important to realize that even though software is divided into assets grouping similar functionality, assets often depend on

other assets. Documentation about the dependencies and care full planning may reduce the complexity, but dependencies still present a problem that the developers and designers need to be aware of.

With a working software product line, organizations have experienced substantial decreases in both cost and time-to-market. Organizations have also experienced an increase of quality in their products[4].

Sony Ericsson Mobile Communications (SEMC) uses a software product line to develop their software. The application software in the phones is developed on top of a common platform that provides the basic functionality for the phones. Different end products include a subset of the application software and it is the process of selecting the assets constituting the desired functionality that will be investigated further.

Currently products are configured by editing a configuration file that specifies all variation points in the software product line. The number of variation points has become very large and a lot of undocumented dependencies exist between them. SEMC is starting to experience difficulties with the variability management since e.g. the outcome of a configuration modification is hard to predict. Another problem is the lack of clear responsibility regarding who should be doing the product configuration.

Traceability between the system requirements and the configured products is another area where problems exist. It is hard to find a connection between the variation points and the general requirements from the product planners. Changes to the functionality aren't always reflected in requirements; instead the developers receive input from testing systems and change the software accordingly[2].

In an initiative to address some of these problems Sony Ericsson Mobile Communication recently started to migrate the existing product configuration to a new, more modular method, of configuring the products. The new suggestion is currently under development and still needs to address some issues. The input this master's thesis will provide with is a proof of concept prototype that uses the new modularized method and some improvement proposals to the process of product configuration.

The goals of this master's thesis will be to investigate the way product configuration is conducted at SEMC and identify the problems involved. The main goal is to develop a prototype of a product configuration tool that remedies the problems found and improves the overall product configuration process. The stakeholder for this prototype is the product configuration managers at the product CM department at SEMC. Another goal is to analyze how the suggested modifications to the product configuration mechanisms, can be used by a tool to further improve the product configuration, including how it can be used together with requirement information to increase the traceability in the configuration process as suggested by Andersson and Nygren[2].

1.2 Research questions

The questions this master's thesis set out to answer are listed below.

1. How are products configured at SEMC?
2. What are the problems with product configuration at SEMC?
3. How can the new configuration method be used to solve the product configuration problems together with a tool?
4. What changes are necessary to get the most out of a product configuration tool?

1.3 Methodology

Different methods were used throughout the master's thesis to get a clear picture of the problem domain regarding product configuration. To initiate the project and introduce ourselves to product configuration and software product lines, articles and papers regarding these subjects were read. This was an adequate method to learn and see different definitions of the nomenclature that appears in theory about software product lines. We then continued the thesis with an investigation about existing, mostly web-based, product configurators, to obtain some ideas about possible features and functionality in the prototype.

The next phase included interviews with employees with different roles within the SEMC-organization. One of the goals with the interviews were to understand who did the product configuration and when. The interviews gave some hints about how a configuration tool could improve the process. An internal course about requirements and processes regarding requirements were taken in parallel with the interviews.

Then the implementation of the product configurator prototype started. The development was carried out in a partially iterative manner.

1.4 Outline

This master's thesis report consists of six more chapters. In chapter 2 an introduction to the basic concepts and theories will be presented to the reader. The purpose of this chapter is to define a common vocabulary and to disambiguate the nomenclature. As part of the analysis, chapter 3 identifies a set of problems concerning software product lines. Chapter 4 presents how Sony Ericsson Mobile Communications work with software product lines and identifies the need for improvements. In chapter 5 a proposal for modification is presented and evaluated. In chapter 6 the development of a product configurator prototype is described. Chapter 7 will summarize and conclude the master's thesis and the improvement proposals.

Chapter 2

Introduction to variant management

This chapter will present the basic concepts and theories used throughout the rest of the report. Definitions of common terms will be provided to avoid misunderstandings. Section 2.1 will describe variability in general and different ways to handle it. A brief description of software product lines and software reusability will be given in section 2.2. To conclude the chapter section 2.3 will describe product configuration in the context of software product lines.

2.1 Variability

In this section the basic concepts of variability, variation point and variation mechanism will be presented and defined.

2.1.1 Variation point

To efficiently derive new variants from the software product line an explicit way to specify and handle the variability must be in place. Some product specific decisions are beneficial to delay so that derived products can be varied. These delayed design decisions is referred to as Variation points. Variability in software systems is realized through Variation Points which identifies one or more locations at which the actual variation will occur [16] [4].

Variability or variant management is one of the more obscure issues in software configuration management. Even though variant management has a high practical relevance, it has not yet gained the level of attention it deserves in the software community [9]. It is often mislabeled as another version control problem.

In an organization using a software product line it is desirable to achieve a high degree of traceability between the requirements and the variation points to

facilitate further development and maintenance. Problems concerning variation points will be further analyzed in section 3.4.

Different definitions of variation points exist and two of them will be presented below. Bachmann et al. defines variation points as "Places in design artifacts where a specific decision has been narrowed to several options but the option to be chosen for a particular system has been left open" [3]. Deelstra et al. defines it as "Places in the design or implementation that identify locations at which variation will occur" [4].

Variability mechanism and binding time are two important concepts concerning variation points. The term binding time refers to the point in a product's lifecycle at which a particular alternative for a variation point is bound to the system. Variability mechanism is the technique that is used to realize the variation points that exists in the software product line [4]. Different mechanisms for this realization will be further discussed below in section 2.1.2.

2.1.2 Variability mechanism

To control the variability and explicitly choose which features (bind variation points) that should be included in a product a well defined variability mechanism must exist. Different alternatives for variability mechanisms exist and two of them, *ifdefs* and *configuration*, will be investigated further in this section.

Ifdefs are used at compile-time and are directives that instructs the preprocessor to include or exclude a block of code. Since the binding is done at compile time, which allows unused variant code to be excluded in the binaries, the overall performance is often increased [13] [2]. A significant drawback of using ifdefs is that the number of potential execution paths tends to explode, due to the combinatorial explosion, resulting in difficulties in maintenance and bug-fixing [13]. Another disadvantage of ifdefs is the scenario that occurs when a new product is introduced, which means having to find and modify all the ifdef statements throughout the entire code base. Mahler has identified two advantages with ifdefs. The first advantage is that "redundancy between different variants of a given component can be almost completely avoided" and the second advantage is that "disc space is used very efficiently" [9]. The most serious disadvantage that Mahler has found is that this variation technique relies on text processing rather than programming language concepts [9].

Configuration is another mechanism used to bind variation points. A new concept called configuration package modules is introduced. These modules points out the actual software that constitute the elements of the products. Products then include these modules to enable certain features and functionality (binding variation point). A module, which is the smallest configurable unit, can have dependencies, like mutual exclusion or hardware dependencies, to other modules imposing constraints on which configurations are valid.

This mechanisms for binding variation points are further discussed in section 4.2.1 when describing how products currently are configured in the software product line at SEMC.

2.2 Software Product Line

A general introduction of software product lines and how they can help organizations achieve a higher degree of reuse will be described in this section.

2.2.1 Reusability

Reusability is a well known concept in computer science and software development. It is the ability to use a segment of source code with slight or no modification to add new functionality. Reusability enables the possibility to build larger things from smaller parts, and being able to identify commonalities among those parts. IEEE defines reusability as: "the degree to which a software module or other work product can be used in more than one computing program or software system." [1].

Organizations using a software product line have one common thing they strive for; they want their developed assets to be reusable to a high extent. Being able to reuse already developed software reduces the time-to-market and the total cost for the product. Also the fact that the software is already well tested and used, resulting in eliminated bugs, is another advantage for the organization.

Deelstra et al. has identified four levels of scope of reuse. The first level is *Standardized infrastructure*. Every new project starts from scratch and the first thing to reuse is they way products are built. This level of reuse standardizes the infrastructure within an organization. Operating system, components such as database management and graphical user interfaces are standardized and reused in future development projects.

The next scope of reuse is: *Platform*. To further increase the reusability within the organization a platform, on top of which the products are built, is maintained. The platform consists of the things described in *standardized infrastructure* and also contains the functionality that is common to all products.

The third scope of reuse is: *Software product line*. Within a software product line, functionality common to a sufficiently large subset of products are reused. Features that are specific to a small set of products are still developed in product specific artefacts. In this scope of reuse variation points are added to accomodate the different needs of the various products.

The last scope of reuse is: *Configurable product family*. When an organization reaches this level almost no product specific development is done. The product family possess a collection of shared artefacts that captures almost all common and different functionality in the products. A new product is derived through reuse of these shared artefacts and no product specific development is required. As soon as this level is reached it is easy to automate product configuration.

2.3 Product configuration

Different alternatives and definitions of product configuration will be presented and defined in this section.

To fully profit from the benefits of software product lines the process of selecting assets should work without effort. This process is generally referred to as *product configuration*. Tu and Skovgaard defines product configuration as "determining a set of components and their relationships, and composing them into a product that satisfies costumer requirement and design constraints" [17]. Kruse ans Bramham defines it as "an arrangement of part, services and assemblies that make up a specific product variant and its associated services" [6].

Two methods considering the initial phase of product configuration will be discussed. A first configuration from the family assets is created during this phase. The input to the initial phases is a subset of the requirements that are managed during the entire process of the product configuration. The two alternative methods, presented by Deelstra et al., that will be analyzed is *assembly* and *configuration selection*.

2.3.1 Assembly

The first stage in the initial product configuration involves the assembly of a subset of the shared product family assets to the initial software product configuration [4]. Deelstra et al. identifies three types of assembly approaches, *construction*, *generation* and *composition*. Since the last one, composition, is a composite of the two other this one is not further analyzed in this report.

Construction

In this approach the initial configuration is created. The configuration is constructed from the product family architecture and shared assets. This approaches starts with the deriving of the product architecture from the product family architecture. The following step is that for each component, select the closest matching component implementation from the component base. The last step in this approach is giving the parameters for each component their respective values [4].

Generation

The generation approach uses a different way, compared to construction, to the initial configuration. In this approach the shared artefacts are modelled in a modelling language instead of implemented in source code. These modeled artefacts constitute the fundamental elements of the initial configuration. And it is a subset of these artefacts that are chosen and constructed to an overall model. It is this model an initial implementation is constructed from.

2.3.2 Configuration selection

Instead of construct or generate the initial software configuration like in the assembly method, configuration selection involves selecting the closets matching existing configuration. The existing configuration consists of an arrangement of components, with the right options and parameters are able to function together.

Old configuration

In the beginning of a new project an *old configuration*, from a previous project, is chosen. The selected old configuration, often from the most recent project, is a complete product implementation containing the latest bug-fixes and functionality.

Reference configuration

Instead of reusing a complete product configuration, only a subset of an old configuration is reused in the forthcoming project. This *reference configuration* constitutes the basis for the new products that are intended to be developed. The reference configuration may be a partial configuration where all the product specific parameter settings are excluded or a complete configuration, including all parameter settings.

Base configuration

The third variant of configuration selection is *base configuration*, which is a partial configuration that acts like a core for a certain group of products. The base configuration must not necessarily be the result from a previous product. It is usually neither an executable application as many parameter settings are left open. One significant difference between old- and reference configuration is that the focus has shifted from reselecting components to adding components to the already predefined set of components in the base configuration.

Configuration selection has one big advantage in comparison to assembly, which is the benefit in terms of efforts saved in selecting and testing. Configuration selection is especially applicable when large systems are developed for customers that have purchased a similar type of system before. This kind of customers often desire new functionality on top of the functionality they ordered for a previous product.

Chapter 3

Problem identification

This chapter presents and organizes a set of problems and impediments that can be found within organizations having a software product line. Four categories are introduced and used throughout this chapter. The first category, which is discussed in section 3.1, is *modularization* which deals with problems that appear when software is divided into modules. In section 3.2 problems concerning *processes* and guidelines are handled. In section 3.3 more practical related issues are analyzed and are included in the third category which is *method*. Issues about creating and maintaining *variation points* are described in section 3.4. The last section in this chapter is section 3.5 which will summarize the problems identified within software product lines.

3.1 Modularization

Modularization of software is as mentioned a common way to achieve reusability. Many of the difficulties with modularization arise from different kinds of dependency issues. Even though strategies to minimize dependencies between software libraries, components, etc. exist, it's hard to completely avoid software coupling. This fact is reflected in, and constraints, the way modules can be used to derive different products. The following problems have been identified in the context of modularization and product configuration:

Software dependencies Modularized software tends to create complex dependencies between the different modules that make up the functionality. It's a complex task to model variability since the assets that constitute the configurable products are diverse and form complex interdependencies [5]. An example of this problem is software modules that have both hardware dependencies and are mutual dependent on another module.

Mutual exclusion In product families it's not uncommon that it exist overlapping sets of functionality. The functionality may be spread over many

modules or configurable by supplying configuration parameters to a specific module. Such overlaps may be the result of options that serve as alternatives, i.e. the functionality given by one alternative can be replaced, to some extent, by the functionality of another. Supplying a value for one of the alternatives effectively excludes the other alternatives. Hence, a specification of the configuration can only specify one of the available alternatives.

Mutually dependent In the same way that configurable alternatives can be mutually exclusive, some alternatives depend on each other in order to work. This means that one of the alternatives cannot be selected without the other one. Perhaps this is an indication that the different alternatives should be merged into one variable option, but more about that later.

Prerequisites Software can be divided in libraries and modules that include each other. The complex mesh of dependencies imposed by this kind of structure falls under the modularization problem - software dependencies. Prerequisites refer to dependencies among different functions of the product being configured. This means that a specific configurable option shouldn't be selectable unless another option is already selected. E.g. it wouldn't make sense to configure the detailed settings of a radio component unless the product was configured to have a radio.

Adaptation When deriving new products from a product family the need for new functionality may arise. The process of developing and adapting the functionality to the product family is referred to as adaptation. The amount of resources allocated to the adaptation process determines if the needed changes will be product specific or integrated with the product family. Product specific adaptation is probably fast but the resources spent on it cannot be reused in other projects. The alternative to product specific development is reactive evolution [4] which aims to integrate the change(s) with the rest of the product family. This is problematic in organizations where time-to-market is an important issue since time constraints will degrade the configurability [4].

Mapping to configuration The stakeholders involved when deriving a new product are many. Depending on their function in an organization they have different views of what product configuration is. E.g. a product planner only cares for the external configuration (variability in the requirements) while a developer is only concerned with the internal configuration (the concrete binding of variation points using a specific variability mechanism). This is problematic because there's seldom a one-to-one correspondence between the external configuration needs and the internal configuration mechanism. Without the ability to unambiguously map customer requirements to actual variation points, the product derivation gets difficult [5] [3].

Scoping of features It can be hard to determine if a feature should be product specific or integrated with the shared assets. It's also difficult to decide who should be responsible for the implementation of the feature [4].

3.2 Process

As mentioned before when we presented the four categories, that each represents a specific scope of concern, the process related issues are supposed to reflect the concerns with product configuration when it comes to processes and guidelines used when developing software. It supposed to include many of the problems and impediments that can arise when deriving a new product from a process related point of view. The majority of the problems that will be described can hopefully be minimized or annihilated by a well established process and well written guidelines.

Lack of methodological support There has been a lot of research about methodological support for designing and implementing shared artifacts in such a way that the actual product derivation should be as painless as possible. [4]. Unfortunately most of the approaches fail to provide substantial supportive guidelines. The consequences, of the lack of methodological support, are that organizations fail to utilize the benefits that software products families imply.

Duplicated errors The presence of errors and bugs in a modern software organization that utilize the idea of product families are inevitable. In the ideal scenario a specific error will only occur once and the effort needed to solve this error will only be used up once. Unfortunately this is not the situation. If a certain error occurs as the result of a particular implicit dependency the importance of externalizing the tacit knowledge becomes obvious. The term externalization is defined as the process of converting tacit knowledge to documented or formalized knowledge [4].

Expert knowledge The need of experts and their knowledge considering different domains are required in a large set of abstraction levels during a product derivation. The different constraints concerning the need of experts are many, e.g., time, costs and the experts availability. As mentioned above, the experts are involved in many of the steps in the product derivation process and because of this they tend to experience a very high workload. This is sometimes experienced as negative by the experts themselves [4].

Tacit knowledge As discussed in Duplicated errors the importance of documenting and formalizing tacit knowledge can be of high interest. This is because of the increasing dependency on experts during product derivation. The drawbacks of this are the vulnerability to loss of knowledge, which may take place if experts leave the organization, resulting in unnecessary repetitive work. The risks of this can be minimized if the tacit

knowledge is divided over multiple experts. However, approaches like this are often expensive and time consuming.

Spread knowledge The knowledge about specific variation points, independently of the abstraction level they belong to, can be widely spread within the organization. Knowledge about low level variation points in source code may be located at the development units while knowledge about variability at product level, which is usually of interest to a CM department, is none of their concern.

Too much documentation Documentation is a vital part of all software development at larger organizations. The drawback of too much documentation for a component is the decreased traceability of relevant information that could be of particular interest when deriving a specific product. Another drawback is the problem related to maintaining the documentation and a consequence of that is that you get out-of-date information, which is worse than no information at all. The quality to quantity ratio concerning documentation requires an acceptable level in order to be effective [4].

Decreased testability The combinatorial nature within product families often negatively affects testability. A large amount of variation points and variants combined with late decisions for the different variation points often makes it impossible to test all combinations (atleast the used ones) during development.[4]

3.3 Method

Regardless of the structure and implementation of variable options in a software product line, there have to be ways in which actual configurations can be specified. This category aims to identify problems related to this kind of work. The following issues have been discovered:

Complex methods Methods used for configuration selection, such as ifdefs and parameterization, tend to become unmanageable as the number of products increase [14]. As the number of configuration options increase, the consequences of a specific configuration choice are hard to predict. A choice that seems good for the moment can possibly have negative consequences in the future [4].

Lack of tool support The methods used for variability modeling and configuration selection often lack tool support to simplify the process. Organizations lack tools and applications that well fit their needs in the derivation process [5]. Without proper tools the process of deriving new products can get time consuming and expensive [12].

Lack of automation Lack of automation could easily be confused with *lack of tool support*, because their differences are somewhat subtle. Lack of

automation means that processes that could be done automatically are currently done manually. Without automation, product configuration can be a hard and error-prone process [4]. The complex dependencies between modules and configuration parameters contribute to a heavy cognitive load on the persons responsible for product configuration. Automation can be hard to incorporate because a large upfront investment may be needed to put the automation into practice. The methods and processes involved with product configuration have to be formalized and well defined.

Parameter dependencies Parameterization is a common way to achieve configurability in product families. As projects get bigger, and the number of products grows, the dependencies between the configuration parameters may get complex and unmanageable. This can lead to a large number of errors and severely complicate the product derivation process [4].

3.4 Variation points

As the software product line evolves and the number of variation points increases a new set of problems, concerning maintenance and organization of variability, arises.

Unmanageable number of variation points Organizations that develop product families have identified the problem that the complexity of variation points increase as they grow in number. The vast number of variation points in some projects, and their complex relations to each other, impose a cognitive load on the individuals performing the setting and selection of variants. As the number of variation points grow these tasks soon become unmanageable. A reason for an organization to have a large number of variation points could be because their product portfolio is magnificent and contains a large number of different products.

Obsolete variation points During the lifetime of a product family, configurable options may become obsolete. This could be due to the fact that new functionality has been added or perhaps the option has become mandatory in every product derived from the family. If obsolete variation points aren't removed, they will contribute to the cognitive complexity the plethora of configurable parameters create [4].

Variation points not organized Without a clear structure of variation points, individuals performing product configuration may have to be concern with variation points that is not relevant to the product being derived [4].

Removal difficulties To reduce the number of variable options, variation points that are no longer used should be removed. This can be difficult since other variation points maybe depend on them. Removing a variation point may thus require reimplementation of the product family [4].

Variation points become invariable As a product family evolves, functionality that only used to be included in some of the derived products may become standard in all products. This eliminates the need for this feature to be a configurable option and the variation point is no longer needed. If this kind of variation points is not removed, many of the previously mentioned issues follow as direct consequences [4].

3.5 Summary

This section will summarize the identified problems. A new large set of problems arises when an organization utilize the benefits of a software product line. This chapter has identified and explained some of these problems. The first initiative to solve a problem is to be aware of its existence, identify and define it. To further ease the approach of solving the typical problems that occurs within software product line, this chapter has established four categories that partitions the problems. The four categories, each containing a similar set of problems are:

- modularization
- process
- method
- variation points

The first category, *modularization* refers to problems that occur when splitting up big systems, whereas the remaining three categories more describes problems and issues related to putting big systems back together in different ways. The variation point's category is especially important because it identifies problems related to variability management like organizing and maintenance of the variability. *Obsolete variation points* and *variation points not organized* have many things in common, but one difference is that the latter one could be the result of that new variation points are introduced from different layers within the organization, e.g, the developers introduce new undocumented variability to ease their software development. While *obsolete variation points* could be the result of remnants from an earlier project. Many of the problems discussed in this category are problems that arises in parallel as the software product line evolves and gets more functionality added.

To solve all of these problems is out of scope for this master's thesis. The remaining chapters of this report will focus on a subset of the identified problems. An example of problems the improvement proposals in chapter 5 will try solve are problems concerning dependencies like mutual exclusion and prerequisites, lack of tool support and a number of problems related to variation points.

Chapter 4

Variability management at Sony Ericsson Mobile Communications

The purpose of this chapter is to discuss how Sony Ericsson Mobile Communications work with software product lines and product configuration. It will also discuss how SEMC experienced some of the problems identified in chapter 3. First a description of the stakeholders in the organization and the interfaces between them will be given in section 4.1. In section 4.2 details about the variability mechanisms used will be discussed. After that a presentation of the requirements system will be given with focus on the creation and management of configuration packages. Section 4.4 will highlight the most important issues that the proposal in chapter 5 will try to solve.

4.1 Organization

To understand the details of product configuration this section will discuss how different stakeholders are concerned with product configuration and identify the interfaces between them.

4.1.1 Stakeholders

To derive a new phone from the product line a wide range of roles needs to be involved, ranging from business people to developers. Roles that are more concerned with the market value of a phone want to create a, somewhat abstract, specification of the desired product. Requirement engineers need to find and specify all the requirements of the products functionality. The developers need to know which of these requirements that should be implemented to create the desired product.

When a product is derived from a software product line (Section 2.3) the people responsible need to know which assets from the product family that should be included. They also need to know how to map the assets to the requirements and the feature specification created by the other roles.

To address the *mapping to configuration* problem described in Section 3.1 the following stakeholders have been taken into account during the construction of the improvement proposal and prototype development:

Product planner Creates concepts of new phones. The product planner is concerned with how competitive a product will be and aims to create new products with features that will attract new customers. In a way, the product planner performs the first product configuration by selecting which features that should be included in a phone. The work done by a product planner is kept on a more abstract level than the requirement engineers, the products are defined in terms of the features they should have.

Requirement engineer Creates and maintains all the requirements of the products. The requirement engineers should analyze the features the product planner wants and find the necessary requirements on a more detailed level. Feedback from the developers about the implementation of the desired modularization may result in changes of the requirements and their dependencies.

Developer The developer role includes all persons that are involved with the implementation of the assets in the software product line. They need to know what to implement and how to structure it in such a way that it's possible to configure the desired products using the variability mechanisms in place.

Product configuration manager Responsible for the binding of variation points. Uses the information about what should be included in a product to select the assets from the software product line using the variability mechanism.

Configuration Manager Responsible for the configuration management within one of the development groups in the organization. The configuration manager receives changes from the developers and delivers them to the product configuration management department.

Software architect Works with the overall design of the assets and makes sure that the different modules can be used together.

Testers/QA Verifies the developed assets by running tests. Supplies important feedback to the developers to inform them if the assets they develop meet the requirements they should implement.

The focus of this master's thesis have been on the work done by the *product configuration managers* and how their work can be simplified. But the changes

proposed in this report also impact, and hopefully improve the work for many of the other roles mentioned.

4.1.2 Interfaces

The roles described above interact with each other in different ways. To clarify and emphasize the communication channels used in the product configuration process the following interfaces have been defined (see Figure 4.1). A lot of other communication channels exist but the ones displayed have been identified as a part of the tool development to understand which information that is, and could be, input to a product configuration tool, and what should be the output. The direction of the interfaces indicates the direction of information related to product configuration.

1. The product configuration managers need to know what should be included in a product in order to bind the variation points and configure the products. It is the product planner that initially created the specification which should be the input to the product configuration.

2. The requirement engineers need to know which features the products should have in order to create the necessary requirements.

3. Mapping between the system requirements and the product configuration. Not currently used, will be discussed further in Chapter 5.

4. Developers deliver the assets to the product configuration managers. The configured and built product can be used by the developers to validate and verify that assets are correctly implemented.

5. The requirements inform the developers of what should be implemented. Chapter 5 will discuss the information the developers need to supply to the requirement engineers.

4.2 Product configuration

This section describes the specific variability mechanisms and product configuration methods used at SEMC. The variability mechanism is currently being changed and the focus of this section will be on the new way of configuring products. More details about the old configuration method can be found in [15] and [11].

SEMC creates a product family for each big project in the company. The product family is used to create many similar phones by exploiting their commonalities. The product family is called a cluster and all the product configuration is done within such a cluster. The cluster is represented by a specific module that defines the different variants in the product family, and contains

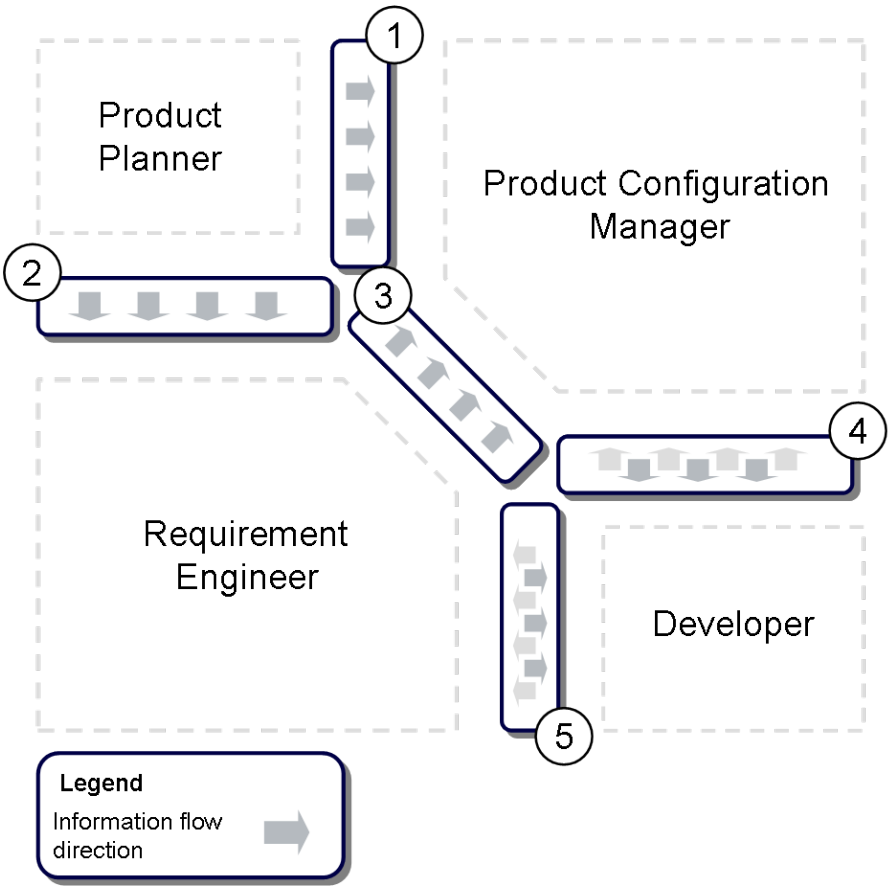

Figure 4.1: Product configuration communication interfaces. Some of the interfaces are all ready in use at SEMC and some have been defined to support the tool development and improvement proposal.

the product configuration. The assets that constitute the products reside in separate modules with their own version history. The configuration in the cluster module specifies which version of the asset modules that should be used, and binds the variation points defined in the modules.

4.2.1 Variability mechanisms

As mentioned above SEMC is currently changing the mechanisms used for product configuration. The old configuration mechanism suffered from a lot of the problems described in Chapter 3 which made it hard to maintain, and the configuration process erroneous in general. To remedy this, the architectures at SEMC has been constructing a new modularized configuration syntax which

will be described in this section.

Modularized product configuration

The new product configuration uses an XML-based syntax to configure the products (see Listing 4.1). The old syntax relied heavily on conditional expressions to make decision about the value of a variation point. This made the configuration hard to understand and change since the value of each variation point could depend on the value of a series of other variation points. The new configuration method is more context sensitive, i.e. instead of asking about what product that is being configured, each product has its own configuration file that contains the configuration specific for that product as depicted in Figure 4.2. Instead of supplying on/off values for every variation point, this configuration method only needs to specify what should be included in the product.

```
1  <product name="Product-1" variant="GENERIC">
2    <configuration-package name="Feature-1">
3    <configuration-package name="Feature-2">
4  </product>
```

Listing 4.1: New configuration syntax. Line 1 declares a new variant of Product 1 called GENERIC. Lines 2-3 specifies which configuration packages that should be included in the variant.

This is made possible by partitioning the old configuration in such a way that the configuration of a product only consists of a specification of which feature the product should have. The detailed configuration of the features has been isolated in the modules implementing the features.

These configurable features have been named *configuration packages* and should be the smallest configurable entity in a product.

In order to achieve this partitioning the software assets needs to be organized in such a way that the inclusion of a single module is possible. This is somewhat problematic because of dependencies between modules or features that require other features to work as described in Section 3.1. The possibility to structure all assets in such a way should be thoroughly investigated but is out of scope for this master's thesis. Methods to deal with possible modularization problems should also be researched.

4.3 Mapping between product configuration and system requirements

In this section the current possibilities to map system requirements to product configurations will be discussed and analyzed. Dhungana et al have identified

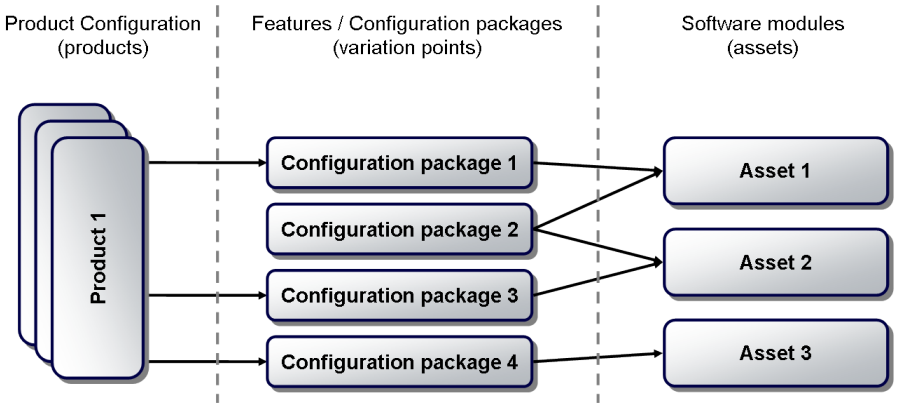

Figure 4.2: Modularized product configuration. A product can only be configured to include configuration packages. The configuration packages should correspond to different features that a phone can have. Many configuration packages can include the same software modules.

[5] that a typical problem with product configuration is the lack of traceability between the decisions taken by customers and sales people and the actual product configuration. It is important to create a mapping between the variation points and their counter parts in other specifications, such as those created by the product planner [3]. The mapping between the requirements and the product configuration is important in order to see whether the built product really implements the requirements it should. Without a clear mapping between them it would be hard to see if a specific feature actually is included in a product, even though it may be implemented and tested. It is also important to have the ability to do the opposite - look at a built product an see which features it includes.

4.3.1 Requirement management

Requirement engineering is an important part of software development. A complete description of the different methods and systems used at SEMC is out of scope for this master's thesis but a simplified description of the parts relevant to product configuration will be given in this section. The discussion about the requirements is meant to investigate the possibilities to map the system requirements with the product configuration. Big parts of the requirement engineering process and the roles involved will be omitted and focus will be put on the way requirements are structured and organized.

Requirement system

All the system requirements at SEMC are stored in a central database controlled by a system that enables the engineers to add dependencies between

requirements, group requirements, generate lists with requirements and perform various other tasks. There are tens of thousands system requirement that together specify every piece of functionality for all the phones in a cluster.

Requirement structure

As mentioned above the requirements systems allow engineers to specify dependencies on a requirement. A requirement can have hardware-, form factor-, operator- and market dependencies. Each of the dependencies can be either inclusive or exclusive, i.e. the dependency is met if the specified dependency is present and not present respectively.

A requirement can be defined to be part of one or more configuration package(s). This is done by simply listing the configuration packages a requirement belongs to in the database post representing the requirement. By querying the database for all requirements belonging to a specific configuration package it is possible to see what a configuration package contains. The (simplified) structure of the system requirements are shown in Figure 4.3.

Currently there are two kinds of configuration packages. The first kind is the configuration packages that are defined by the requirements engineers in the requirements systems. These configuration packages may actually exist in two places, once in the requirements system where they are declared, and once more if they are implemented using the new XML-based syntax. The other kind of configuration packages are only implemented using the XML-syntax, and doesn't have a counter part in the requirements system. The difference between the two is that only configuration packages declared in the requirements system can define dependencies that can be used by e.g. a tool. This kind (requirement system compliant) of configuration packages are the preferred sort but currently the other kind is needed in the process to migrate the configuration to the new syntax. The term configuration package will be used to refer to any of the two and is intended to be thought of as the logical unit that represents a variation point in the software product line.

The difference between the configuration packages in the requirements system and the XML-implemented is that the ones in the requirements system are their logical representation and the XML-files provide their implementation, i.e. it is the declarations in the XML-files that states which modules etc. that should be included in a specific configuration package. And it is the XML-versions that are used when products are being configured.

The system requirement database also contains a specification of which configuration packages that should be included in every product of the cluster and the hardware attributes of each product. This is the result from the initial configuration done by the product planner together with the requirements engineers.

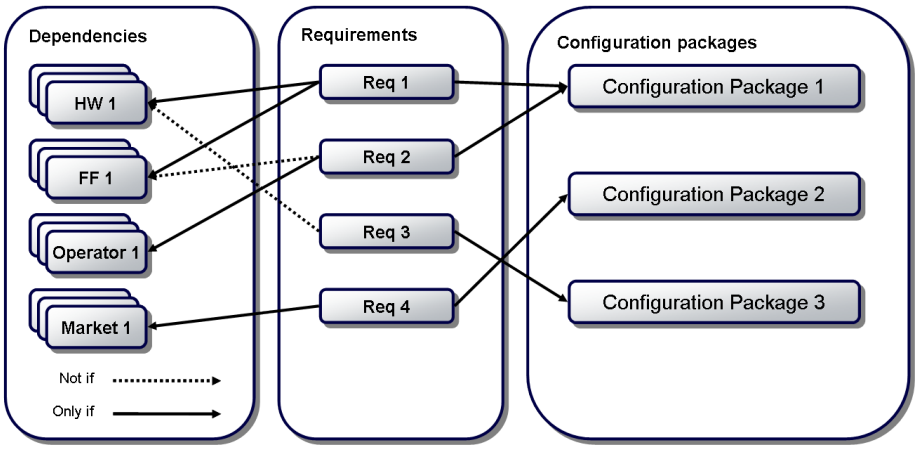

Figure 4.3: Requirement system structure

4.3.2 Mapping

The product configuration at SEMC currently doesn't enable any mapping between the requirements and the product configuration [2]. The new modularized syntax proposal has addressed this issue by suggesting that the configuration packages used in the XML-syntax should correspond with configuration packages defined in the requirement system. Most of the product derivation is done by copying old configurations and changing the configuration after receiving feedback from the testing organization.

4.4 Analysis

The analysis of the results and findings in the previous sections will be presented in this section.

4.4.1 Variability

The different roles identified in this chapter are all involved in product configuration to some extent. However, their tasks are very different and the kinds of problems they have differ widely since they encounter product configuration on different levels of abstraction. The identification of how the needs for a product configuration tool differed between the roles was an important factor when the proposal and tool was created.

A product planner needs a simple way to specify the features of a new product without caring too much about the underlying variability mechanisms and architecture. The ideal situation would be if the product planner could create a clear specification that could be directly transformed or mapped with the variability mechanisms. Due to practical limitations such as dependencies

between modules, component prices, etc. the product planner might have to revise the initial configuration of a product, but those changes should be easily reflected in the configuration.

The product configuration manager should be more concerned with how the specification supplied by the product planner can be transformed into the variability mechanism used. This is the kind of information that could be transfered via interface 1 (see Figure 4.1). The current product configuration at SEMC lacks this kind of mapping.

On the least abstract level, the developers need to assist in the product configuration process. Since it is the developers that create the assets that should be configured they need to be aware of the constraints and limitations the assets they develop need to adhere to. The developers need to know about the desired variability in order to create the corresponding variation points. A problem at SEMC is that there's lacking control of when and how variability should be introduced in the software product line. Most of the variability is actually introduced by the developers. When variability is introduced by the developers interface 4 is used for product configuration, rather than delivery of the assets that should be configured by the product configuration managers. A goal of the proposal presented in this master's thesis is to change the product configuration from being done in a bottom-up manner, into a more top-down process.

The organization of the system requirements with configuration packages and dependencies aren't currently used in the product configuration. The creation of the configuration packages is an important part of the variability management since a goal was to only use the configuration packages as variation points. The grouping of system requirements into configuration packages is done by the requirement engineers together with the product planners. Interface 2 represents this work, i.e. the requirements on the variability come from the product planner and should be implemented, logically, by the requirement engineers. Communication from the developers to the requirement engineers via interface 5 may also be needed since the developers possess the knowledge about dependencies between modules and hardware etc. that needs to be reflected in the configuration packages.

4.4.2 Configuration packages and variation points

As suggested by both the software architects at SEMC and Andersson et al[2] the variation points used by the new modularized configuration syntax should correspond with the configuration packages defined by the requirement engineers. During the analysis of the requirement and configuration package structure a couple of problems have been identified.

Products from the software product line should only be configured by selecting which of the available configuration to include. In order to achieve this all of the functionality must be available in such packages. It is also important that the packages are specific enough to allow a high degree of configurability, but at the same time general enough to not make configuration management

too complex.

Currently SEMC configures the products using both the old variability mechanisms and the new. Only a few modules have been migrated to the new syntax, and the product configuration still heavily relies on the old configuration files. The mapping between the configuration packages and the configuration is only suggested and a thorough investigation if the existing configuration packages are enough to enable the desired degree and variability haven't been performed.

As described in Section 4.3 the configuration packages are made out of system requirements. These requirements may, or may not have dependencies. This means that the configuration packages effectively inherits all dependencies from the included requirements. The way the configuration packages currently are constructed have some interesting consequences because of this.

Variable configuration packages Assuming it is desired that the product configuration match with the specification of configuration package available in the requirements system, the content of a configuration package is variable, i.e. depending on which product currently inspected, the content of an included configuration package may change. This is the result from requirement dependencies propagated to the configuration packages. According to the configuration specification two products with different hardware configuration can include the same configuration package, which may depend on a hardware feature available in only one of the products. This kind of variability within the configuration packages are depicted in Figure 4.4.

This however, is only a problem if the configuration is allowed to be done in such a way, that a product missing a hardware attribute required by a configuration package, can include that configuration package anyway. If the new modularized configuration syntax is used, this kind of variability within configuration packages would make configuration very confusing since it would be hard to tell which functionality the end product would get.

Contradicting dependencies An easy solution to the problem above would be to simply enforce all the dependencies, i.e. it wouldn't be possible to include a configuration package in a product if some dependency isn't met. As it turns out, some configuration packages currently include requirements that directly contradict each other. Some of the configuration packages defined in the requirements system have requirements that both states that a specific hardware attributes are required and specifies that it isn't allowed. If all dependencies have to be met this configuration package could hence never be included in any product.

None existing dependencies Another issue with the current structure of the configuration packages is that some requirements depend on hardware attributes that aren't listed in the product configuration specification. If

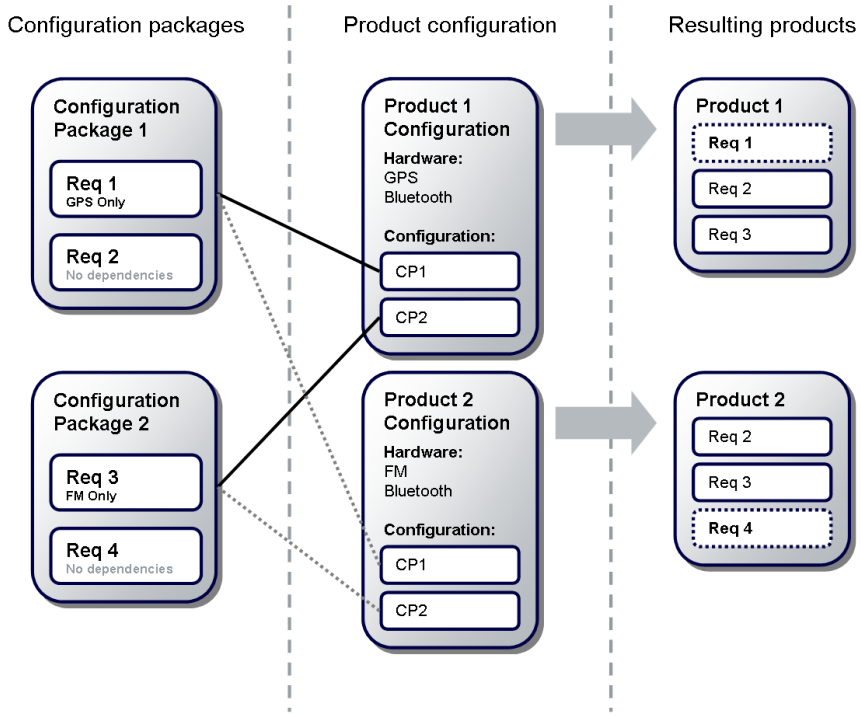

Figure 4.4: Configuration package variability. Example of two products with different hardware traits, configured using identical configurations. If variability within configuration packages is allowed, based on e.g. hardware, the same configurations can result in different end products.

it isn't possible to specify that any product have the needed hardware attribute, the package can never be used.

All of these problems have been identified from a product configuration point of view. In other words, the problem with the configuration package variability and contradicting dependencies may only be a problem when the mapping between the configuration packages and the variability mechanism is done. The ability to specify dependencies the way it's currently done lets the requirement engineers specify differences among different products in an easy and intuitive way. Most of the problems arise when you strictly need to enforce the dependencies and need to know what should be included in a product or not. Different ways to make the configuration more dynamic is subject to further investigation.

4.4.3 Responsibilities

One of the problems at SEMC seems to be the lack of clear responsibilities of who should do what. There is no clear specification that states who is responsible for the maintenance of the variability in the software product line. Variability is introduces by both product configuration managers and developers. The introduction of new variation points is done without any impact analysis on the configuration.

If the configuration packages defined in the requirements system are going be the variation points in the software product line, it is important to implement a well defined process of how they should be maintained, both in respect to product configuration and requirement engineering. It is also important that it is clear which roles that are responsible for the different maintenance tasks.

In order to address this problem the benefits of the needed changes have to out weigh the cost, e.g. time it would take to change the way the different roles are working. During the interviews that was conducted as a part of this thesis work, a common complaint about the current process was the lack of clear responsibilities.

Chapter 5

Proposals

The following section will present a proposal for changes that SEMC could perform in order to improve the product configuration process. The proposal is divided into a general process suggestion, that suggests how existing initiatives within SEMC can be used together to improve the process, and a rule set, that the variation points in the software product line need to conform to. Section 5.1 will present the process modification and the rules are presented in section 5.2.

5.1 Top-down configuration

In this section we will present a proposal of how the product configuration process could be changed to make it more manageable. As discussed in Section 4.4.1 the product configuration is currently mostly done by the developers, which makes it grow from the bottom. The problem with this is that it gets hard to maintain the configuration. In order to control the the variability it is important to control when, why and by who variability should be introduced in the software product line. Without sufficient amount of control over the configurations many of the problems identified in Section 3.4 will appear, such as *obsolete variation points*, *unmanageable number of variation points* and *variation points not organized*.

To make the product configuration more manageable it should be done top-down, i.e. the product configuration managers should configure the products according to the specifications created by the product planners and requirement engineers, forcing the developers to develop assets that enables and works with this variability.

5.1.1 Responsibilities

One common problem identified at SEMC is the lack of clear responsibilities. The following descriptions of each role's responsibilities are supposed to be in-

corporated together with the other suggestions in this proposal and a tool such as the prototype described in chapter 6.

Project planner Creates the variation points, i.e. constructs the configuration packages together with the requirement engineers. Introduction of new variability and removal of old, should be based on decisions taken by the product planner.

Requirement engineer Responsible for maintenance of the configuration packages. They should assure that the proper dependencies are defined and that the needed variability is reflected in the available configuration packages. Since the configuration packages are initially declared in the requirements systems, and other versions of the package such as the implementations in XML should reflect these ones, the role that knows most about the requirement system and the reason for the creation of the package should also maintain it.

Product configuration manager Bind the variation points specified by the requirement engineers and product planner according to the product configuration specification. The product configuration manager should also update the product configuration when a developer group have made changes to the content of a configuration package and delivered the changes.

Developer Develop the assets and configure configuration packages to contain the correct modules. The maintenance responsibility for every configuration package should be assigned to a group of developers. The developers will make sure that the configuration package implements the requirements it is supposed to, and deliver configuration package changes to the product configuration managers.

5.1.2 Connection between requirements and configuration

The *product configuration specification* should fully reflect the configuration of the products. The product configuration specification consists of a product/feature matrix that lists all product variants in one dimension, and all configuration packages in the other. A variant is configured by simply marking the configuration packages that should be included in the column corresponding with that variant.

The product configuration specification is maintained in the requirements system and should be created by the product planner together with the requirement engineers.

When the product configuration managers configure a product this document should be the specification of what should be included.

With proper tool support the mapping between the product configuration specification and the actual variation point binding can be automated and consistency checked.

5.1.3 Non functional configuration

It is important to enforce the dependencies available in the requirements system. Configuration packages may depend on certain hardware features in order to function, or should only be included if the product is to be released to a specific market. To enforce these dependencies the tools used needs to be aware of the specific traits of the product currently being configured.

We suggest that all non functional configuration, i.e. hardware-, operator-, form factor- and market configuration, should be done with configuration packages representing that specific hardware attribute, form factor, etc.

By representing this kind of configuration this way e.g. hardware dependencies can be modelled the same way as dependencies between ordinary configuration packages.

5.1.4 Dependency representation

The purpose of the dependencies is to guide and help the roles performing the product configuration to construct valid configurations. The dependency information available from the requirements systems can be used both to prevent the construction of illegal configurations and validate existing configurations.

We suggest that the tools used to create the product configuration specification and the tools used for product configuration should query the requirement system about the dependency information. By doing this the users of the tools can e.g. get valuable feedback about which hardware components that are needed, and information about which prerequisites a configuration package may have.

In order to use the information from such queries it is important that the suggestions from Section 5.1.2 are implemented.

5.2 Variation point structure and rules

This part of the proposal will present suggestions on methods that could be used to solve some of the problems with the configuration package structure identified and define a set of rules that the configuration packages need to conform to.

5.2.1 Method suggestions

To achieve the mapping between the system requirements and the product configuration some of the problems identified in Section 4.4 needs to be fixed. In order to use the existing variability mechanisms the variability within the configuration packages needs to be removed. The configuration package variability that currently exists is caused by features that have optional functionality and requirements that serve as alternative implementations. These features need to be split up into configuration packages that can be included in a product without the need for variation.

We have created two alternative strategies that could be used to achieve this kind of split.

Method 1

One approach to solve the problems with conflicting dependencies and variability within configuration packages is to create one configuration package for each desired variant of the configuration package as shown in Figure 5.1.

The upper part of the figure illustrates a configuration package that includes two requirements with contradicting dependencies. In this case two almost identical configuration packages could be created only differing with respect to the contradicting requirements. A common reason for this kind of contradictions is because the requirements implement different alternatives of the same functionality.

The lower part of the figure shows a configuration package containing requirements that are supposed to be optional, i.e. included in the product if it has touch screen or GPS. In this case a configuration package representing each configuration alternative could be created and the product configured with the appropriate one.

One of the advantages with this method is its simplicity and easy implementation. A disadvantage is that the configuration packages will require double maintenance when the common parts are changed.

Method 2

The second method suggestion is more complex than method 1. Instead of creating multiple configuration packages with mostly the same content, this method works by only extracting the affected part into configuration packages of their own. In the example with contradicting dependencies (see Figure 5.2) the extraction of requirements Z and Y creates dependencies between the originating configuration package and the extracted ones. The dependencies are marked with **1** and **2** in the figure. Dependency **1** is necessary if the extracted requirements constitute a required part of the functionality in *Feature A*, such as the source of the positioning data in a map application. Dependency **2** is necessary since *Feature A Z* and *Feature A Y* probably wont work, or even make sense, without *Feature A'*.

Dependency **1** is problematic because *Feature A'* doesn't depend on both of the extracted configuration packages, only one of them. In order to model this dependency a mechanism to describe this would be needed. One way could be to declare a common interface that the alternative implementations could be a part of, and specify a dependency to this interface in *Feature A'*, marked with **3** in the figure.

In the case of optional features, as shown in the bottom part of Figure 5.2, only the dependencies between the extracted packages and *Feature A'* are needed, marked with **4**.

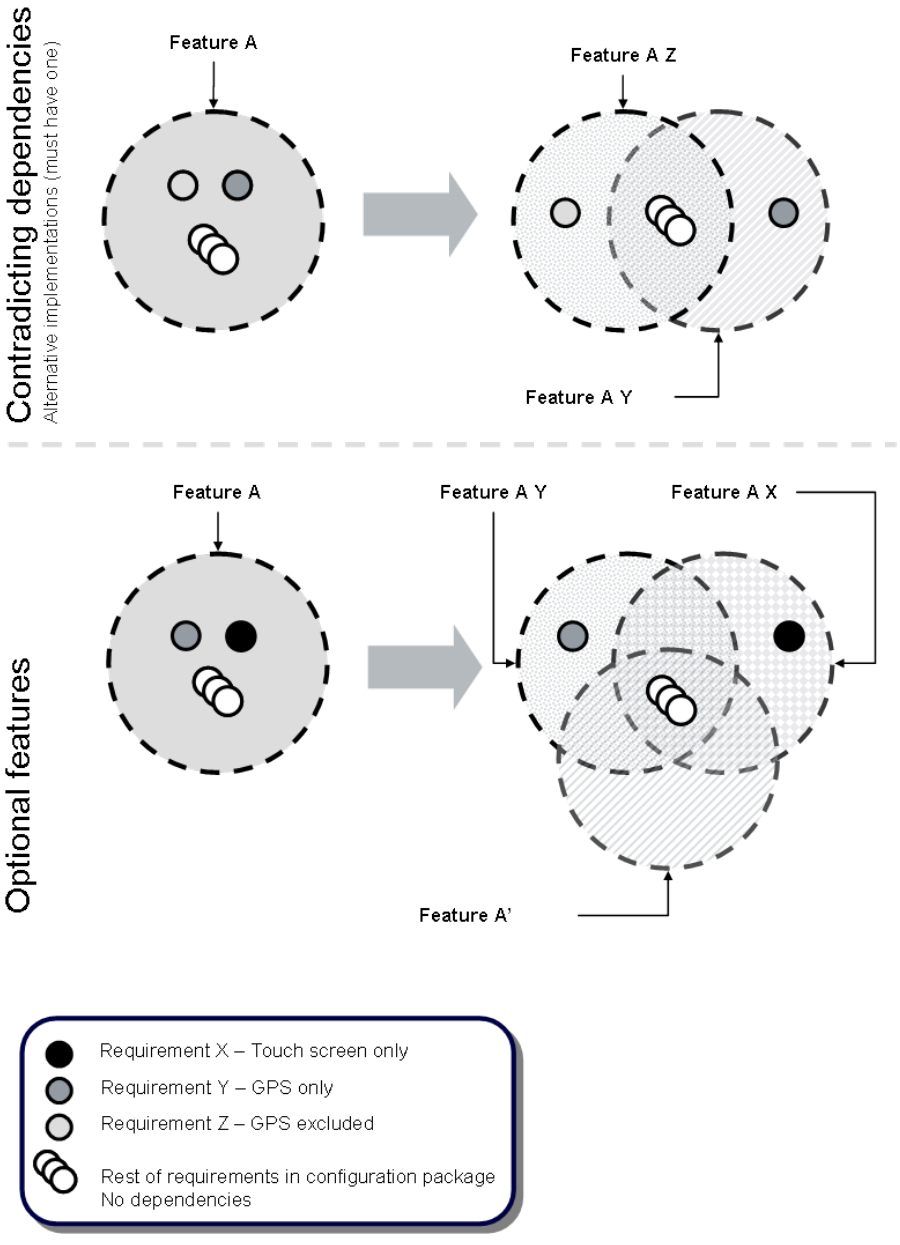

Figure 5.1: Method 1. The figure illustrates how a configuration package can be split into many configuration packages to resolve contradicting dependency issues or remove variability from a configuration package.

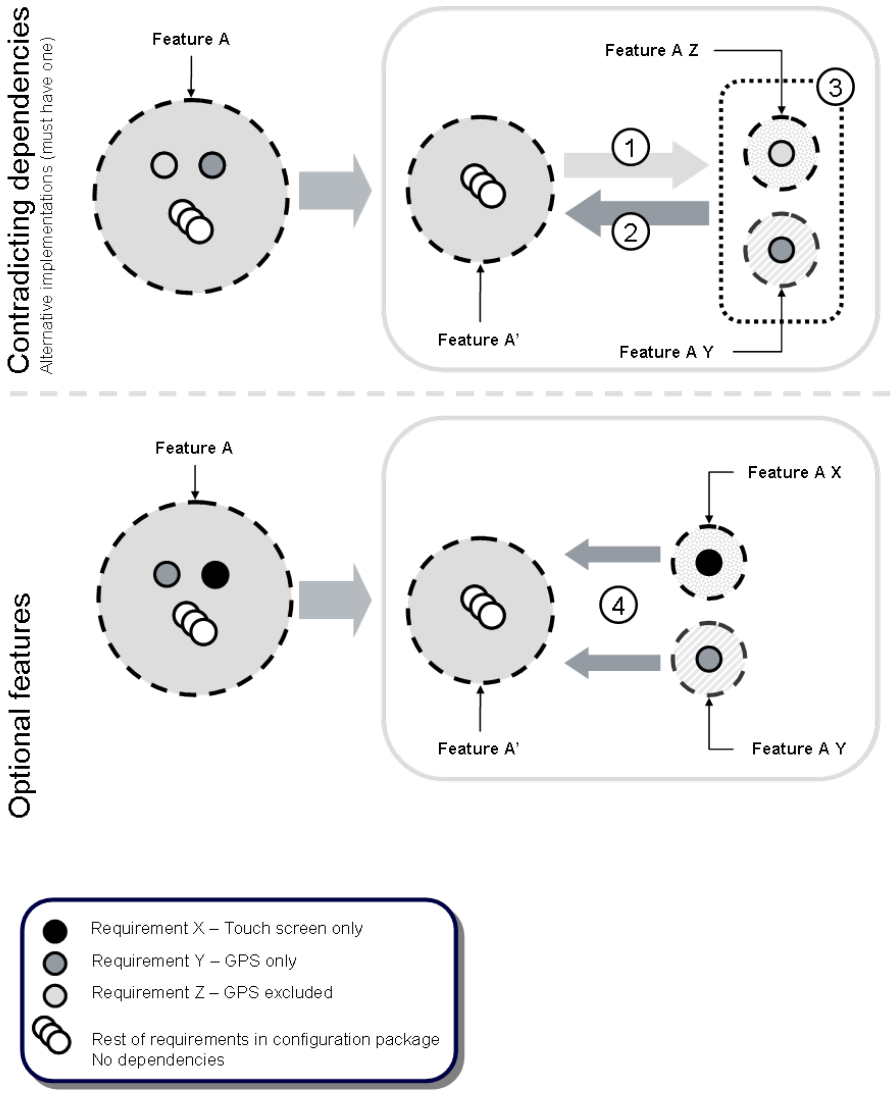

Figure 5.2: Extraction of configuration packages.

The advantage with this method is that the need for double maintenance would be eliminated, but the complexity of the configuration would increase because of the need for dependencies and interface declarations.

5.2.2 Rules

In order to map the requirements to the product configuration, using the existing variability mechanism, the following rules about the structure and creation of

configuration packages needs to be obeyed.

- Forbid configuration packages with contradicting dependencies.
- Enforce dependencies when a configuration package is included in a product.
- The product's hardware configuration should match with the hardware configuration in the product configuration specification.

The purpose of the rules is to proactively prevent the identified problems with the structure of the configuration packages. The first rule should prevent the creation of configuration packages that is unusable using the current variability mechanism. The second rule is supposed to prevent invalid configurations. It is easier to stop an error from being made, than finding what's causing it afterwards. In order to strictly enforce all dependencies the configuration packages cannot depend on things that don't exist.

5.3 Discussion

The contribution of this proposal will be summarized and evaluated in this section.

The three main parts of the proposal are clear responsibilities, usage of the variability mechanism and structure of the variation points. The clarification of responsibilities is needed to make the product configuration more top-down rather than bottom-up.

The arrows in Figure 5.3 represent different tasks. The arrow marked with **1** is the process of constructing the configuration packages and create the product configuration specification. This work should be done by the product planners together with the requirement engineers.

Arrow number **2** indicates the clear specification, viz. the product configuration specification, that the product configuration managers should use as input to the configuration. The configuration packages used in the specification should be implemented by the developers, indicated by arrow **3**. The developers also need to provide feedback to the requirement engineers to help them construct the configuration packages in such a way that they are implementable, hence arrow **3** is bidirectional.

The developers, that implement the building blocks in the product configuration, should deliver the configuration packages to the product configuration mangers (arrow **4**). We suggest that these releases are freeze labels to modules that implement one or more configuration package(s) using the new modularized syntax. The product configuration managers can use these labels to add new, or change existing, configuration packages. This is done by adding or changing an entry in the XML-file that defines which modules that supply configuration packages.

Finally the product configuration managers should use the delivered configuration packages and the product configuration specification to configure and build the products (arrow **5**).

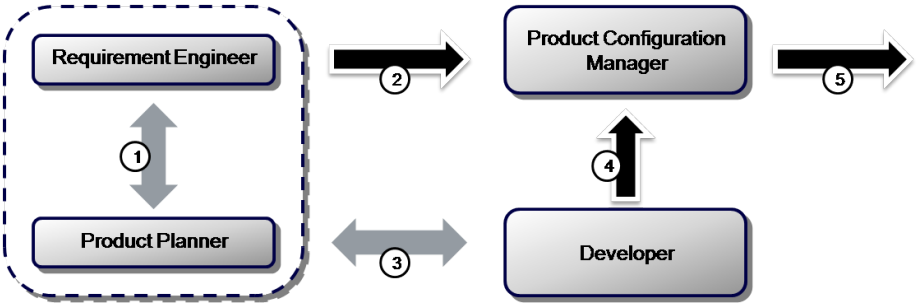

Figure 5.3: Configuration process overview. The black arrows indicate tasks that are supported by the tool prototype described in Chapter 6.

Even though the work involved with arrows **1** and **3** isn't directly supported by the tool, it can still be used in both cases to get an overview of the current state of affairs, e.g. check wether it is possible to remove a configuration package without breaking any dependencies.

The second part of the proposal concerned the mechanisms used to represent the variability. We suggest that all product configuration should be done using configuration packages, including hardware, operator and market configuration. All configuration packages should be available in the requirements system and dependencies between them should be enforced when the products are configured. This should be done by using a tool that can communicate with the requirements system as well as interpret the XML-based configuration syntax.

Our suggestions of how to deal with the structure of the variation points, i.e. the configuration packages, contained a suggestion of how the existing problems can be dealt with by splitting up the configuration packages. We suggest that *method one* should be used because of its simplicity and since it doesn't require any changes to the mechanisms already in place. As a part of the proposal we also defined a set of rules that the configuration packages and product configuration needs to obey.

One of the reasons for SEMC to migrate the product configuration to the XML-syntax is because of the unmanageable number of variation points in their software product line. During the work we looked into what's called formal concept analysis (see [7][8]) and how it could be used to model the configuration and identify e.g. obsolete variation points. We realize that it would not be easy to partition the entire old configuration into the new smaller configuration packages but formal concept analysis could probably be used to ease the process. Due to its complexity we decided to focus on how the configuration can be done using the new syntax and how the dependencies can be used and visualized. The formal concept analysis used a product-feature matrix to construct a lattice

consisting of all the different products and features. Before we decided not to use the formal concept analysis we also investigated different ways to parse the old configuration files[1]. Since we decided not to use the formal concept analysis the result from this was also discarded.

[1] http://research.cs.queensu.ca/~thurston/ragel/
http://flex.sourceforge.net/
https://javacc.dev.java.net/

Chapter 6

Tool development

One of the main goals of the master's thesis was to develop a proof of concept prototype that visualize dependencies and facilitates the product configuration. The tool was intended to give the people at the product CM department new insights about how product configuration could be done to make their every day work easier. The general needs concerning the prototype is analyzed in section 6.1. Issues about variations regarding the tool is in section 6.2. The main goals that constituted a foundation for the development are presented in section 6.3. The actual implementation and development of the prototype is discussed in section 6.4. Some points and ideas about how further development of the prototype could be performed, and some hints about additional investigations concerning product configuration in software product lines, are located in section 6.5.

6.1 General need

This section will deal with the general needs that the developed prototype should conform to.

Before the actual implementation of the prototype started, a research of existing product configuration tools was carried out. The purpose of this research was to obtain some innovative and pioneering ideas regarding product configuration. The results from this investigation constituted a basis for further analysis of problems involved with product configuration and their solutions. A wide variety of tools were examined and evaluated according to the guidelines and criteria's defined below. To demonstrate the information that was captured during the research a real example is given.

Product Online car builder

Indented user(s) Buyers

Description Web based tool that let customers configure/build a car online.

Usage

1. Select a car model. All available cars are listed with the corresponding price information.
2. Select an engine. All engines available for the selected car model are listed.
3. The user can see the cars standard equipment and select custom accessories. The customization is done by selecting checkboxes next to the description of the item.
4. Select color of the car and its interior.
5. (Generate a cost estimate)

Advantages

- When the user picks an item that conflicts with another item, all choices that conflict with that choice is indicated to be "impossible".
- The order in which items are selected is natural and intuitive.
- The cost estimate is updated as the user selects items.
- If the user clicks and item in the list a short description of that item is given.

Disadvantages

- The lists are small and hard to read.
- No clear overview of what is selected.
- When a custom choice generates impossible configuration, it is hard to track what option that was disabled.
- The tools get slow and unresponsive when a lot of checkboxes are selected.

Even though this example is from the car industry, many of the features and functionality that are described are also of importance to a software product configuration tool.

Research results

The results from the research was not used to a large extent but provided some helpful insights into what features are of importance. The investigated product configurators were online, web-based car and computer configurators. Some example of features that were discovered and later taken into account when the prototype development started was to visualize incompatible configurations dynamically with colors, information about both software and hardware dependencies, divide similar configurations with tabs and the possibility to show further information about software assets. Some examples of features that not

were implemented in the prototype are details button for each specific part and cost estimates.

What also was discussed during the research phase was the interest to choose a base configuration when constructing a new product, and then augment it with further functionality depending on the system requirements. This way of deriving a product adhere much to the assembly method called, *base configuration*, that is analyzed in section 2.3.2.

Use cases

The next step in the tool development process was to create use cases that described typically scenarios during product configuration. The use cases were divided into two separate categories: *Product* and *Configuration packages*, where each of these two categories was further divided into sub-categories. The category Product was divided into product set-up, product configuration, maintain configuration and maintain hierarchical structure. Configuration packages was divided into one sub-category called create configuration package. Each use case was given a unique ID, a descriptive name, intended actors, assumptions and the none-functional requirements. An example of a use case, concerning *product set-up*, that constituted a foundation for the actual prototype is given below.

Use case Set up new product variant from scratch.

Description Set up a new product from scratch with a name and list of included features.

Actors Product CM, Product Planner.

Assumptions

- All the information, such as name, and included features, should be available to Product CM.
- Configuration is feature-based.
- Including none-implemented functionality does not break anything.

None-functional It should not be possible to create a product with the same name and variant as an existing product.

6.2 Tool variations

Different variants and variations of a product configurator will be described in this section. Two types of product configuration tools were discussed before a final decision, about these two implementation alternatives, was taken. The first candidate used the old syntax and configuration method. This variability mechanism is briefly described in section 4.2. As mentioned before, this mechanism, currently used in the projects running at SEMC, is supposed to be superseded by the new, XML-based, modularized initiative. So one advantage

of implementing a tool using the old syntax, compared to a tool using the new initiatives, is that the tool hopefully could be used within a short time frame. On the other hand, a disadvantage is that the tool would soon become old and obsolete because of the changes to future projects. Also, two previous master's thesis, [15] and [11] have already been investigating these method and proposed some improvement suggestions.

The other alternative of product configurator would be a tool based on the new modularized way of configuring products. This XML-based syntax is presented and discussed in section 4.2.1. One interesting feature of this new method is the possibility to define abstract products. The abstract products are a way to avoid unwanted redundancy and group similar functionality into base products. Because of these abstract products and the inheritance structure, different scenarios related to adding and removing configuration packages appeared. Two scenarios regarding product configuration, called *generalize* and *specialize*, will be analyzed in the next two sections.

Generalize

Generalization is a scenario that may occur when a configuration package is added to a product variant. Picture 6.1 visualizes an inheritance tree and a simplified version of the generalize-scenario.

Three products exist in this cluster. One abstract product named *Base product* and two concrete product variants named *Product 1* and *Product 2*. Base product has two local configuration packages, CP1 and CP2. The two product variants have one configuration package each - CP3 and CP4. As the picture depicts, the both concrete products inherits from the abstract product thus they both got two inherited, non local, configuration packages.

The scenario that number 1 in the picture refers to is what happens if CP3 is added to Product 2. In the next transition (2), CP3 is moved to Base product because all of its children possessed the configuration package. As a result of this, Product 2 no longer contains any local configuration packages, instead all three are inherited.

The drawback of this solution is the side effect that products (Base product and its children) that explicitly wasn't involved had it contents changed (The intentions were to change only one product, but the result of that change restructured an entire sub-tree). In order to achieve more control of the product configuration, the result of this scenario should have ended after the first arrow, i.e., CP3 would never be moved to the abstract product.

Specialize

The specialize-scenario is the somewhat reverse scenario of generalize. This is a scenario that may occur when configuration packages are removed from products. Picture 6.2 visualizes an inheritance tree and a simplified version of the specialize-scenario.

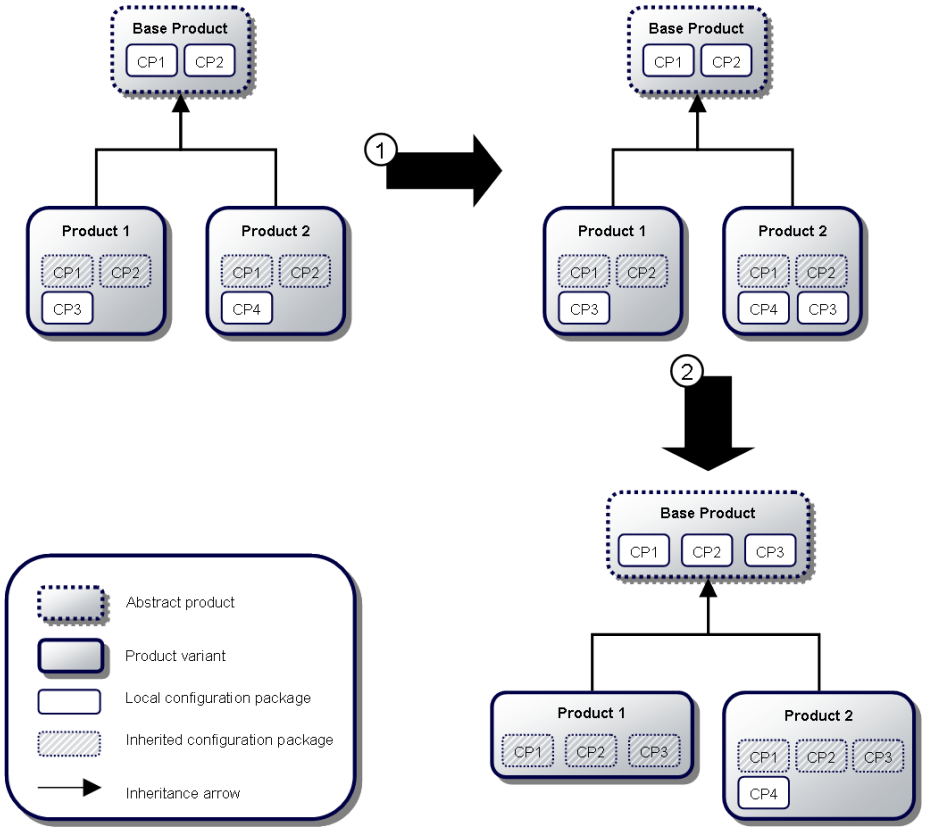

Figure 6.1: Generalize

The initial cluster, with one abstract and two concrete variants, is identical to the one used for generalization.

Number 1 in picture 6.2 shows what happens if CP1, which is an inherited configuration package, is removed from Product 1. The package is removed from the abstract product, resulting in that Product 2 ends up in an invalid product state because of the lack of CP1. To solve this, the lost package must be re-added to all children of the abstract product, except the one the configuration package was removed from, to make the products valid again (Like in generalize, an entire sub-tree was reorganized when the intentions were to change the content of one product). In the scenario in picture 6.2 only Product 2 was involved in the re-adding of configuration package.

The lack of control discussed earlier in *generalize* is a problem here also because of the implicitly restructuring within the inheritance tree. It is not possible to end the process in number 2, because that would leave Product 2 in the invalid product state. A constraint that makes is impossible to remove none-local configuration packages is a more appropriate approach to this scenario.

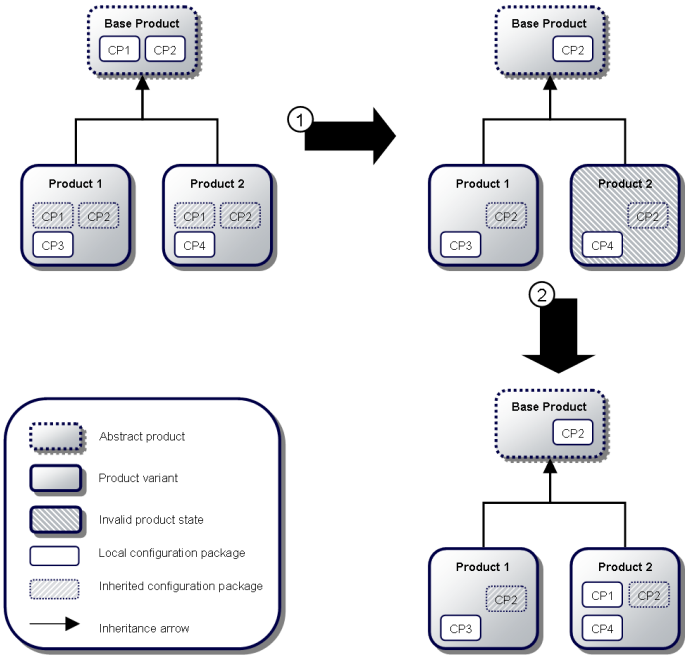

Figure 6.2: Specialize

Thus the only way to remove a configuration package is if the specific package was earlier added to the product, making it product specific and local.

6.3 Tool goals

In what way a tool could facilitate the actual product configuration will be this sections main topic. The new modularized initiative, described in section 4.2.1 was used for the developed prototype.

The decision concerning that the prototype should be a stand alone application instead of an application integrated in their current build environment was taken with some valuable input from the thesis supervisor and documentation from previous master's thesis. The supervisor, who carried out his master's thesis two years earlier at SEMC, developed a dynamic analysis tool into the current build environment. The experience he gained from that project was that the build environment was too complex and the time spent reading source code and documentation was too much for a 20 weeks master's thesis.

A subset of the most important features and requirements of the prototype are listed below.

User interface The prototype should consist of a rich, cognitive and intuitive graphical user interface.

Cluster configuration The possibility to specify a cluster to work with, instead of a single product variant, should exist.

Dependencies The application should visualize the different dependencies that can exist on configuration packages. Dependencies like mutual exclusion, prerequisites, form factor- and hardware dependencies.

Inheritance structure A perspicuous reflection of the inheritance structure that exists within the cluster.

Configuration package Visualize if a configuration package is local or inherited.

Enforce constraints A configuration package with a prerequisite or another type of dependency that is not fulfilled should not be addable.

New products Both abstract and concrete product variants should be constructible form scratch. The possibility to create a product using a *base configuration* should also exist. For a detailed description of base configuration see section 2.3.2.

Configuration Possibility to add and remove configuration packages from a product. Specify hardware, form factor, operator and market.

Updating labels The prototype should facilitate the process of updating which labels that should be used for a specific product.

Interfaces

The different interfaces, discussed in section 4.1.2 and visualized in picture 4.1, within the SEMC-organization was also taken into account when the goals for the prototype were arranged.

In an ideal scenario much of the product configuration, analyzed in chapter 4, would be eliminated. This configuration, currently done by product configuration managers and developers, would instead be done in an earlier stage. The preferred way would be if the product planners could use a tool to specify the initial configuration and then deliver this to the product configuration managers, this communication channel is depicted as interface 1 in picture 4.1.

Another desirable goal for the tool was related to interface 4 in picture 4.1. Input from the product configuration managers informed us that it is sometimes advantageous if the developers have the possibility to construct a tentative product configuration for test purpose during the development of certain functionality. This tentative configuration could be created by a product configuration tool.

6.4 Tool implementation

Implementation specific issues and problems that occurred during the implementation of the prototype will be presented in this section.

Before the implementation presentation some new technical terms needs to be introduced.

RCP RCP is an abbreviation for Eclipse Rich Client Platform (RCP). RCP is the minimal set of plug-ins needed to build a rich client application in Eclipse. John Wiegand, Eclipse Platform PMC Lead says "We did not explicitly set out to create the Eclipse Rich Client Platform (RCP). The Eclipse mission was to create a universal tool platforman open, extensible integrated development environment (IDE)" [10].

JAXB JAXB is an abbreviation for Java Architecture for XML Binding. JAXB can make it easier, compared to ordinary XML-parsers, to access XML documents from applications written in the Java programming language.

The prototype was developed in Java using JAXB and RCP. JAXB enabled easy read and write access to the XML configuration files.

Dependency information

The central database which is discussed in section 4.3 was used to access the domain knowledge about the software dependencies. Instead of working directly against the requirement system database, a mockup database was used. The reason for this was the lack of time dedicated to this thesis. The mockup database was simply a replica of the database containing all dependency information regarding the configuration packages. The drawback of this was that the mockup database had to be rebuilt as soon as the real database was updated.

Software architecture

Figure 6.3 depicts a simplified version of the software architecture. The input to the prototype is the configuration files that contain the current product variants from the software product line and their current configurations. As discussed in the section above, the prototype is using a replica of the requirement system database to acquire the domain knowledge regarding dependencies between the configuration packages. The output from the tool is same configuration files that acted as input.

Implementation problems

Some minor problems are inevitable when developing a tool of this size. One of the bigger problems that occurred during the prototype development was the problem concerning hardware and form factor configuration. The first proposal to hardware and form factor configuration involved a new tag that was

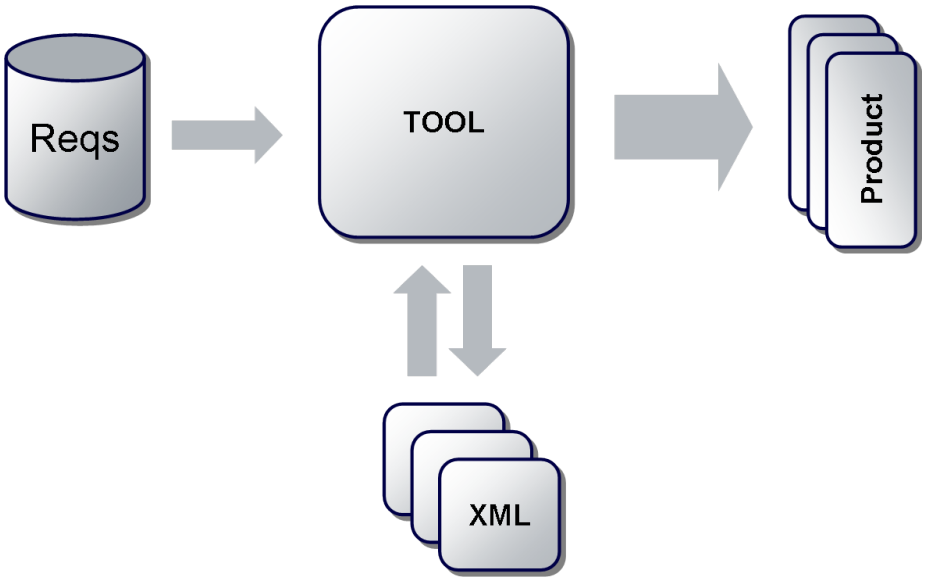

Figure 6.3: Tool architecture overview.

introduced to the XML-files. These changes to the XML scheme and the XML configuration files were unwanted and a new way of configuring hardware and form factor had to be invented.

The method that later on was used had very much in common to the ordinary software configuration. Each different hardware and form factor had their own configuration package introduced. Section 5.1 contains a more detailed and verbose information about this proposal.

6.5 Future work

What further improvements and features that could be added to the prototype will be discussed in this section of the report.

If a new development team were supposed to continue the development of the prototype we would recommend investigating the opportunities to incorporate a connection to the central database. Using this method instead of the mockup database would be the preferred way because of two reasons. The first reason is to avoid the manually export of the central database to the mockup database. The other reason is the fact that the tool will, in realtime, receive updated information about the dependencies.

Another subject open for further investigation is the methods called generalize and specialize. These two features were never implemented because of the lack of control discussed earlier. A deeper research about these two methods and the different alternatives that arises could be of interest. E.g instead of

automatically do the generalization, implement an executable feature that do the restructuring (optimization) within the inheritance tree.

Linux kernel

A research about the linux kernel and how the kernel is configured could be a interesting topic for future work. What have the Linux community accomplished to achieve such a popular configuration method? Is the kernel configuration file, which is kept in the .config file in the top directory of the kernel source tree, an alternative configuration method that SEMC would benefit by.

Formal concept analysis

A method that was briefly investigated is a method called Formal Concept Analysis (FCA). FCA is mathematical method that provides a way to identify meaningful groupings of objects [7]. The input to this method is a matrix that specifies a set of objects and there properties, often called attributes. In this context the objects are product variant and the properties are variable features.

The *Product Configuration Specification*, discussed in section 5.1 and 4.4, would probably be an appropriate candidate as the input matrix. The method constructs a concept lattice including all concepts. For more information about the lattice and concepts consult [7]. This lattice contains information about the objects and their relationship to each other. The reason for using this technique is to analyze the usage of variable features in product configurations. The analysis will hopefully find a way of restructuring and simplifying the provided variability of the components.

Chapter 7

Conclusion

The goal of this thesis was to investigate how Sony Ericsson Mobile Communications performs product configuration in their software product line and how they could improve the process with the help of a tool. We have studied the variability mechanisms currently in place at SEMC to get a clear picture of the difficulties with product configuration. This has been done by reading documents, performing interviews and taking courses.

Our studies have resulted in a proposal that consists of changes that SEMC needs to perform and a prototype product configuration tool that utilizes these changes and shows how product configuration could be done more easily.

We have identified that products currently are configured bottom-up at SEMC. This means that it is the developers that both introduce the variation points and binds them. By allowing this it is very hard to manage the software product line since variability is more or less introduced without control.

Instead of letting the developers create variation points, we suggest that the variability should be defined by the requirement engineers together with the product planners. This has all ready been suggested by Andersson and Nygren[2], but we introduce an additional set of rules that the configuration packages, defined by the requirements organization, must conform to. The purpose of these rules is to more easily be able to use the dependency information in the configuration packages to control that invalid configurations cannot be created. We have also presented methods that could be used to change the existing configuration packages in such a way that they will conform to our rules.

If the available variability in the software product line is defined by the configuration packages, we think that the product configuration will be more top-down. It will also provide a good amount of traceability between the configured products and their requirements. We think it is important that the product configuration is performed in a top-down way to make the software product line more manageable.

A big part of the work done is the development of a product configurator prototype. The goals with the prototype were to show how product configura-

tion could be more simpler and show how the proposed changes could be used to accomplish it. The tool uses the new modularized configuration syntax and enables the desired connection between the configuration and requirements systems. By having the tool enforce configuration package dependencies and clearly visualize product configurations we think we've showed how configuration could be more effective. The tool also includes features to create new products from scratch or using an existing product as reference. It is also possible to modify and manage the underlying configuration structure to further simplify and streamline the configuration process.

The subjects we want to emphasize to summarize this conclusion are the importance of the proposed set of rules concerning product configuration and to show how a tool, that e.g. prohibits invalid configurations, could simplify the process.

Bibliography

[1] Institute of Electrical and Electronics Engineers. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. 1990.

[2] J. Andersson and S. Nygren. Managing variability requierments and variation points for software product lines. Master's thesis, Lund Institute of Technology, 2008.

[3] F. Bachmann, M. Goedicke, J. Leite, R. Nord, K. Pohl, B. Ramesh, and A. Vilbig. A Meta-model for Representing Variability in Product Family Development. *Software Product-family Engineering: 5th International Workshop, PFE 2003, Siena, Italy, November 4-6, 2003: Revised Papers*, 2004.

[4] S. Deelstra, M. Sinnema, and J. Bosch. Product derivation in software product families: a case study. *The Journal of Systems & Software*, 74(2):173–194, 2005.

[5] D. Dhungana, P. Grünbacher, and R. Rabiser. DecisionKing: A Flexible and Extensible Tool for Integrated Variability Modeling. *First International Workshop on Variability Modelling of Software-intensive Systems*, 2007.

[6] G. Kruse and J. Bramham. You choose [product configuration software]. *Manufacturing Engineer*, 82(4):34–37, 2003.

[7] F. Loesch and E. Ploedereder. Optimization of Variability in Software Product Lines. *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 151–162, 2007.

[8] F. Loesch and E. Ploedereder. Restructuring Variability in Software Product Lines using Concept Analysis of Product Configurations. *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR07), Amsterdam, Netherlands, March*, 2007.

[9] A Mahler. Variants: Keeping things together and telling them apart. *John Wiley & Sons, Inc. New York, NY, USA*, 1995.

[10] J. McAffer and Lemieux J-M. *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java^{TM} Applications*. Addison Wesley Professional, 2005.

[11] B. Pileryd and A. Hellström. Controlling the variant explosion - enforcing stability in highly configurable large scale software. Master's thesis, Lund Institute of Technology, 2005.

[12] R. Rabiser and D. Dhungana. Integrated Support for Product Configuration and Requirements Engineering in Product Derivation. *Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on*, pages 219–228, 2007.

[13] Svahnberg. Supporting Software Architecture Evolution - Architecture Selection and Variability. *PhD thesis, Blekinge Institute of Technology, 2003.*

[14] M. Svahnberg and J. Bosch. Issues Concerning Variability in Software Product Lines. *Software Architectures for Product Families: International Workshop IW-SAPF-3, Las Palmas de Gran Canaria, Spain, March 15-17, 2000: Proceedings*, 2000.

[15] S. Thörngren and V. Karadzic. Modeling dependencies in dynamic software configurations. Master's thesis, Lund Institute of Technology, 2007.

[16] D.L. Webber and H. Gomaa. Modeling variability in software product lines with the variation point model. *Science of Computer Programming*, 53(3):305–331, 2004.

[17] B. Yu and HJ Skovgaard. A configuration tool to increase product competitiveness. *Intelligent Systems and Their Applications, IEEE [see also IEEE Intelligent Systems]*, 13(4):34–41, 1998.