Managing Variability Requirements and Variation Points for Software Product Lines

June 2008

Jonas Andersson
jonas.t.andersson@gmail.com

Samuel Nygren
samuel.nygren@gmail.com

Master of Science thesis
Lund University, Engineering Faculty
Department of Computer Science

Supervisors: Pär Åsfält, Sony Ericsson Mobile Communications AB
Krzysztof Wnuk, Lund University
Examinator: Björn Regnell, Lund University
Abstract

This thesis presents an evaluation of the current product derivation process for the product line approach at Sony Ericsson Mobile Communications. The thesis examines the weaknesses in the current derivation process and the gaps between the requirements and the configuration of products.

During the evaluation of the current process it was found that products were configured in an iterative way, based on change requests and bug reports, rather than using specifications. The major problems that were identified involved fragmented product specifications, a large number of variation points and unclear responsibilities for the product configuration activities.

To be able to solve these problems, improvements were proposed involving a new structure connecting the configuration to the initial requirements while separating the configuration into two levels of granularity. This was done to assure that products are configured according to requirements, in a more appropriate granularity, that responsibilities are clear and that the configuration can be verified. To handle the lack of product specifications we also proposed the use of a main product specification, which is based on the same entities as the configuration, to be used throughout the organization.

The evaluation of the proposal showed that there were no major obstacles applying the proposed changes to the current configuration interface. Surveys were used to get opinions and comments, which showed that stakeholders were overall positive to an introduction of the proposal.

Keywords: Software Product Lines, Variation points, Variability requirements, Configuration Management, Product Derivation, Product families
Acknowledgments

We would like to thank Björn Regnell and Krzysztof Wnuk at Lund Institute of Technology, Department of Computer Science for supervision and reviews of the report. A special thank to Pär Åsfält at Sony Ericsson Mobile Communications, for continuous feedback, help and excellent supervision. We also would like to thank the Product-CM department and others who have contributed with important information for this case study.
## Contents

1 Background and Methodology ................................................................. 7
   1.1 Background and goals ................................................................. 7
   1.2 Research Questions and Methodology .......................................... 8
      1.2.1 Research Questions .............................................................. 8
      1.2.2 Case study methodology ....................................................... 8
      1.2.3 Evaluation of current process for variability management .... 8

2 Introduction to the subject ..................................................................... 12
   2.1 Variability ......................................................................................... 12
      2.1.1 Identifying variability ............................................................... 13
      2.1.2 Constraining variability ........................................................... 13
      2.1.3 Implementing variability ........................................................... 14
      2.1.4 Managing the Variability ........................................................ 18
   2.2 Software Product Lines .................................................................... 18
      2.2.1 Initiating a software product line ............................................. 19
      2.2.2 Development process ............................................................... 21
      2.2.3 Variability representation ......................................................... 23
      2.2.4 Requirements engineering ....................................................... 23
      2.2.5 Configuration Management ..................................................... 26
      2.2.6 System architecture ................................................................. 28
      2.2.7 Software Evolution ................................................................. 29
   2.3 Other case studies ........................................................................... 29
      2.3.1 Celsius Tech Systems ................................................................. 29
      2.3.2 Mobile browsers at Nokia .......................................................... 30
      2.3.3 Bosch Gasoline Systems ............................................................ 31
      2.3.4 Cummins Engine Inc ................................................................. 32
      2.3.5 Ricoh Image Memory Handler .................................................. 32
      2.3.6 Philips Consumer Electronics .................................................... 33
      2.3.7 Siemens Medical Solutions ...................................................... 34
      2.3.8 Salion Inc. .................................................................................. 34
      2.3.9 Thales nederland B.V. ............................................................... 35

3 Case Study ............................................................................................... 36
   3.1 Introduction ......................................................................................... 36
   3.2 Evaluation of current process for variability management ............. 37
      3.2.1 Software Product Line ............................................................... 37
      3.2.2 Overview of the current development process ....................... 37
      3.2.3 Organization .............................................................................. 38
5.1.4 Evaluation with CM-strategists .................................. 90
5.2 Evaluation Analysis ................................................... 93

6 Future work and Conclusions .......................................... 94
   6.1 Future work at SEMC ............................................... 94
   6.2 Future research .................................................... 94
   6.3 Conclusions ......................................................... 95

A Evaluation forms ....................................................... 99
   A.1 Evaluation form - Improvement proposal .......................... 100
   A.2 Evaluation form - CM-Strategists ................................. 102
Chapter 1

Background and Methodology

1.1 Background and goals

The goals of this master thesis is to evaluate the current process of how new products and variants are configured and identify its weaknesses. The final goal will then be to formulate an improvement proposal of the current product configuration process, made in line with ambitions of Sony Ericsson Mobile Communications (SEMC).

In the last decades software have entered more and more areas and have become an integrated and important part of most peoples everyday life. When new physical functionality is replaced with software the time and cost for the effort spent on software development can reach vast proportions. Therefore, software reuse becomes more and more important. Partly to decrease the development cost and time spent on testing, and partly because it often result in more stable systems [FvDL07]. Today, reuse of software is one of the main techniques used when building systems that share common software and sets of components. One way to do this is by using a software product line. This is done by letting a set of software systems share common sets of features to be able to support specific needs and to satisfy different market segments in a more efficient way. A product line also defines constraints in the way in which systems are developed, to be able to maintain its evolution over time.

Today, integration among different software systems is very common. Therefore, there is a higher need for flexibility when designing and developing variability structures that are helpful when creating variants of software systems. Here, variability means the points in the software where the behavior of the functionality can be changed. Normally, these points are controlled through an interface that provides access to alter the behavior according to the requirements.

The goals of a product line approach is mainly to reduce the costs by increasing the productivity and the software quality through software reuse. This have been proved successful in several organizations [Bos00] [FvDL07]. Here, software evolution is very common and this need to be taken into account when several systems are sharing the same functionality.

In the beginning of a project, new technologies and requirements are nor-
mally first used in one or a few products. As the market matures, some requirements might eventually be a part of all products. To avoid variability explosion and reduce the maintenance effort of old variability points, a process for managing variability requirements and removing obsolete variation points is necessary.

At SEMC, a software product line is in place and working satisfying, but there is a need to increase knowledge in how products are really configured today. Also, the number of variability points in the product configuration at SEMC have escalated and the number of obsolete variation points is increasing for every new project. These are some of the underlying reasons behind the outset of this master thesis.

1.2 Research Questions and Methodology

1.2.1 Research Questions

1. How are variability requirements and variability points managed in Software Product Lines in a real industrial example from the mobile phones domain?

2. What are the problems with managing variability requirements and product derivation?

3. What improvements can be made in managing variability for Software Product Lines?

1.2.2 Case study methodology

The thesis will be carried out as a case study, where the current process will be described in detail using techniques as document studying and interviews to collect the necessary information.

A case study describes a specific case for a specific purpose, which means that the outcome of the case study should not be considered to be generalizable [MHR06]. Although we do not have any statistical evidence, the findings from a case study is probably relevant for other organizations in the same field.

The design of a case study is flexible, meaning that the questions, techniques and direction of the study may change. This allows for gathering information in a flexible way, adapting the research to be as effective as possible. [MHR06]

The information collected during a case study is most often qualitative. In our case we have worked primarily with qualitative data, although when evaluating the proposal we used a survey containing both qualitative and quantitative questions.

1.2.3 Evaluation of current process for variability management

The evaluation of the current process was performed by studying documents, taking internal courses and performing interviews.
The documentation was acquired from SEMC’s internal document management tools and consisted of process descriptions, specifications and design documents. The supervisor pointed out essential documents and other documents were found by following references and by exploring the document database using the obtained knowledge. The requirement management system was also explored to understand how variability is defined at requirement level.

A company internal course on requirement management for platform requirements was taken to get an overview of the process. The learning material from the course has also been used.

Key personnel was interviewed to get different perspectives on how variability is achieved and new products are configured.

Persons involved in process improvement for the requirement process was interviewed in connection to the internal course. They were also consulted during the exploration of the requirement management system.

By analyzing SEMC’s product configuration interface, the amount of variation for different development groups were established. One group with a large amount of product variation and one group with a small amount was selected for further investigation. The selection of roles for interviewing was focused on covering the whole process of variability and involved product planners, requirements engineers, requirement coordinators, developers and system testers. The interviewed persons were selected randomly in some cases, and based on recommendations from earlier interviews in other cases. Some of the persons that were asked to participate proposed colleges in their place, with the motivation that they were unfamiliar with the area. 27 persons, with different roles where interviewed. The distribution is shown in figure 1.1.

<table>
<thead>
<tr>
<th>Role</th>
<th>Dev Team 1</th>
<th>Dev Team 2</th>
<th>Overlapping Roles</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Planner</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Req. Engineer</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>Req. Coordinator</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Process Manager</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>SW Architect</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SW Developer</td>
<td>2</td>
<td>5</td>
<td>-</td>
<td>7</td>
</tr>
<tr>
<td>Test Engineer</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Configuration Manager</td>
<td>-</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>4</strong></td>
<td><strong>10</strong></td>
<td><strong>13</strong></td>
<td><strong>27</strong></td>
</tr>
</tbody>
</table>

Figure 1.1: Distribution of the interviews with the different roles.

The interviews was semi-structured, meaning that questions was formulated in advance and used during the interview, but resulting questions was asked too. This allowed for the interview to change direction depending on the interviewee’s answers, giving a deeper understanding of the persons perspective. The questions were adapted for the different roles and the progress of the thesis, meaning that early interviews were very general and later interviews had more specific questions. The purpose of this was to concentrate on the indistinct areas.

During the interviews, which were approximately one hour, both of the authors took notes continuously. After the interview an informal summary based
on the notes was written. During the writing of the summary discrepancies between the authors notes was considered and questions needed to sort these out was sent to the interviewee along with the summary and a model of how the interviewee perceive the process of how variability and new products are created. The interviewee approved the summary and the model, in some cases after minor changes. This data was then formulated as different perspectives, both on the current way of working in section 3.2 and on the problem situation in section 3.3.

After approximately 15-20 interviews, less and less new information occurred. Therefore, after having interviewed 27 persons, it was decided that our overview of the current process was satisfying to be able to proceed with the improvement proposal part of the thesis work.

Sample questions

The following questions are samples of those used for the interviews. As mentioned earlier the different roles had an adapted set of questions. The questions listed here are a mixture of questions used for different roles, with the purpose to give the reader an understanding of the granularity and range of the interviews.

- Briefly describe your role.
- How are new products created? Which documents are used?
- Are you familiar with the Product Configuration Specification and the concept of Configuration Packages and Configuration Dependencies?
- How do you achieve the differentiation between the products? How is this communicated to the developers? Is the differentiation documented?
- Who is performing the actual configuration of the products, i.e. the binding of the variability? What documentation is this work based on?
- Are Configuration Packages and Configuration Dependencies used to represent variation?
- When the Configuration Packages are created, are they based on the existing requirement mass?
- How are customer specific requirements communicated?
- How are the developers informed that variability is needed in the platform?
- How have you been involved in the configuration parameters and variation points in the code?
- Do you experience any problems with how configuration parameters and variation points are handled? Do you have any suggestions on how this can be managed in a different way?
- Do you have any contact with personnel responsible for the sub products? Do they ask you about variability and configuration?
Pros and Cons

The way in which people were chosen can lead to insufficient results. When choosing which people to interview randomly, one have to bare in mind that their overview might not give a full picture of the current process. On the one hand, by getting recommendations of which people to interview there is a risk that the selection can become to narrow or subjective. This is though a good way to get in contact with people possessing the desirable knowledge you are looking for. Of course, the amount of people being interviewed from each role is in this case not enough to be able to present the result in a proper statistical way. The goal must always be to cover as broad range of practice as possible. However, the addition of this thesis does not allow for covering the full range of practice.

As a part of the selection process, questions concerning each practice area have been asked to at least two different groups or roles, according to Clements [CN02a]. Further, Clements states that the interviews should be performed by product line experts. This criteria is not met, though studies on the subject was made before interviewing phase began.

When asking people, it is important that the questions is not threatening or intimidated to the person being interviewed [CN02a]. This have been taken into account by making sure not to call the person to account.

There are some potential threats that might impact the analysis of the studies. The result can be influenced by continuous communication with the supervisor at SEMC. Recommendations by people from previous interviews might also narrow the received perspectives. There is also a possibility that some concerned stakeholders might have been overlooked in different parts of the case study.
Chapter 2

Introduction to the subject

Software of today are most often produced as a family where different variants are available. These variants can be specialized for different environments or for different products offering less or more functionality. For example, an embedded program might exist in different variants for different CPU’s, a program calculating salary will have different variants for different tax laws and even the simplest program is often available in one debug and one release variant [Bab86]. Another way to use variants is to broaden the market by adapting the products to suite the requirements of new markets. Being able to sell more copies, the high development costs of a complex software system can be distributed among the customers. [Mah95]

The commonalities and differences between variants must be captured so that they remain identical in ways they should be identical, and differ in ways they should differ. If this is done and the variants are managed and controlled, the productivity can be greatly increased and unnecessary work can be eliminated. If the variants are not managed correctly, we will soon see the effects of the double maintenance problem described by Babich. [Bab86]

In order to manage and control the variants in an efficient way, we need to store the parts of the program that are common and the parts that differ so that double maintenance is avoided. This also allows for reuse of the parts between the different variants or products, which is one of the goals with software product lines [Sva03] which will be mentioned in detail later. Different techniques for implementing and organizing the variability in the software will also be presented. Related processes like requirements engineering, configuration management and system architecture will be described in the context of software product lines.

2.1 Variability

Variants and variant management are one of the most complex tasks in software configuration management. It is often seen as a version control problem, but the problem is not as well understood or supported by tools as version control. To understand what variability is and what the actual problems are, we need to define the different elements of variability. Unfortunately, there is no common definition for what a variant is. Some of the definitions are too technical and
restrictive, while other definitions are too general to be helpful. [Mah95]

Variants could be seen as parallel versions, adding an extra dimension to the version tree of a version controlled item [Mah95]. Babich represents this by coequal alternatives of a module, residing in parallel under a module interface. Each of these alternatives has its own version tree representing that a module has both variants and revisions, reflecting the two dimensions [Bab86]. While parallel versions are supported by most version control systems by the use of branches, this concept is most often too simple to efficiently manage variants [Mah95].

The parts of the software that differs between the products in the product family are said to be variant features. This means that the feature can have alternative implementations, variant implementations, which can be chosen to create different products. The set of implementations that are available to choose from is called a variant population. Each point in the software where different variant implementations from a variant population can be chosen for a variant feature is called a variation point. Depending on which realization technique that is used, a variation point can be the point where a class is chosen to be used or where code fragments are chosen to be run. A configuration where a variant implementation has been chosen for each variation point results in a variant or product.

The work of identifying where the products differ is important to be able to decide where the variability point is to be introduced and what technique to use. [Sva03]

2.1.1 Identifying variability

The first step in achieving variability is to identify where it is likely to occur. Svahnberg proposes the use of feature graphs where the differences in features between the products can be easily identified [Sva03]. A feature graph is a representation of the requirements specification where sets of requirements are grouped into features. This can be done iteratively to create a hierarchy reaching from feature bundles down to some appropriate granularity. From this graph it is easy to identify where the variants differ and at what stage in the implementation it is most suitable to place the variability point associated with the variability feature.

2.1.2 Constraining variability

By examining the evolution of other systems the architecture of the system can be adapted to cope with the variability that can be foreseen. However, the architecture should not be too flexible since every variation point introduces new maintenance and reduces the understandability of the software. To develop an architecture that provides just enough flexibility, the life cycle of the variant feature should be considered. [Sva03] has identified the following stages for a variant feature:

**Identified** The variant feature has been identified as a feature that may vary between products.

**Implicit** The variation exists as a concept but has not yet been implemented.
Introduced The technique for realizing the variability has been chosen and the variation point has been implemented.

Populated The different variants that can be chosen for the variation point has been implemented and are ready to be chosen. The population can take place anytime from design to runtime.

Bound One of the populated variants is chosen to be used at the variation point. This means that the variation point is replaced by the chosen variant, making it impossible to chose another variant without going back in the process. This can be done when the architecture is composed using a CM-tool, when the software is compiled, linked, started or even during runtime.

The binding of a variation point can be divided into External binding, meaning that the system is not aware of the variation point and does not need to consider the variability, and Internal binding, meaning that the system needs to be aware of the variation point and choose which variant feature to populate with. This means that external binding is done outside of the system, by a software architect or developer. This is typically done before compilation of the system. Internal binding is however done by the system, by e.g. analyzing the value of a configuration variable and choosing the correct functionality using a simple if-statement [Sva03]. The more technical aspects of the techniques will be covered in the next section.

Different ways of implementing the variability mechanism puts the life cycle stages of the variant feature (identified, implicit, introduced, populated and bound) in different stages of the software’s life cycle (designed, compiled, running). For example, the variant feature may be bound at compile time (external binding) or runtime (internal binding) resulting in software that can vary in a later stage of its life cycle. Introducing a variant feature at an early stage and binding it at a late stage results in a life cycle where the software can vary over a long time. This means that designers, developers and testers need to manage all variations, making the process very costly. It is therefore preferable to bind the variant feature as early as possible, keeping the time where the variation point is open as short as possible. This is illustrated in figure 2.1. [Sva03] [GBS01]

2.1.3 Implementing variability

A straight forward way to introduce a new variant is to simply copy the baseline and modify it to accommodate the requirements of the variant. Soon we will see the effects of the double maintenance problem, as bug fixes and other changes needs to be applied to both the baseline and the variant. If the implementations of these changes are not identical, the variant and the baseline will start to diverge in ways they were not supposed to, leaving the product in an inconsistent state. For this reason it is important to choose the correct technique for the variability, allowing the variant implementations to share the parts that are common. [Mah95] [Bab86]

There exist many techniques for implementing variability and storing variants. Depending on which technique that is chosen for the variant feature the time for introduction, population and binding differs. The decision of when to introduce a variant feature should be concerned with the size of the software
Figure 2.1: Number of possible systems that needs to be managed, depending on when the variation is bound. [GBS01]

entities, the number of variation points and the cost of maintaining the variant feature [Sva03]. The chosen technique also affects the representation of the variants, and the parts that are shared between the variants [Mah95].

A large set of variation points for a variant feature affects the maintenance and understandability of the design and source code in a negative way. Therefore, instead of working with lines of code, it is preferable to gather the variating parts into classes or components with common interfaces, allowing for a single variation point for the variant feature. [Sva03]

The following sections describe some of the realization techniques proposed by [Sva03], [Bab86] and [Mah95].

**Variant segregation**

Variant segregation imply that the different variant implementations are separated into different components, classes or files. This is the organization of variants that is suggested by most version control systems. It can be achieved either by using the hierarchical file system structure to separate the variants in different folders under the variant feature, or by using the branching capabilities available in most version control systems. The former representation is simple, intuitive and works well with ordinary version control but it offers less transparency when managing the variants and it is not obvious how common code is shared between the variants. The latter alternative stores all code in a single archive and uses branches to represent the differences between the variant implementations. This approach is not as simple to grasp, but it allows for common code to be shared from the main branch to the variant implementation branches. [Mah95]

The segregation of the variants can lead to two major problems. The first problem is the double maintenance problem, occurring when modifications are made to the different variant implementations. If the modifications apply to
the shared parts of the code these must be propagated to the other variant implementations, but it is not always obvious if the changes should be made to common code or to variant specific code, with risk to introduce variant implementation into shared code. The second problem of variant segregation is multiple variance. A component may vary in more than one dimension, offering e.g. different functionality in one dimension and different environment support in another dimension. The organization of the variant implementations is no longer obvious in the hierarchy, and implementing every combination leads to a combinatorial explosion. [Bab86] [Mah95]

**Architecture Reorganization** The architecture is reorganized using techniques like Architecture Description Languages (ADL). This is used when the variants have the same set of components, but the data flow between the components differ. The binding is usually done in the design phase. [Sva03]

**Variant Architecture Component** Variants are implemented as components and stored in different files or branches in the repository. The correct variant of a component is chosen using a configuration management tool. The binding is done when the code is checked out from the repository [Sva03]. Because the variants are stored in different files or branches the commonalities between the variant implementations are not shared, resulting in the double maintenance problem [Bab86]. The version control system ClearCase has a powerful concept with branches applied to directories, allowing the variant implementations to be represented as branches in the directories version tree. Being able to branch in a subsystem allows for easier separation of common and specific source code for the implementations, while preserving the transparency while working with a single variant implementation. ClearCase also has a dynamic tracking of the dependencies between components, by logging of the files requested by the compiler when building the variant. [Mah95]

**Optional Architecture Component** Like the previous technique, but there is also a null-component available making the functionalities in the variant component optional.

**Binary Replacement - Linker Directives** The correct binary for the variant is chosen during the linking process. [Sva03]

**Binary Replacement - Physical** The correct binary is physically replaced. This is possible when using i.e. dynamically linked libraries. Binding is done before the software is started allowing modifications after delivery. [Sva03]

**Infrastructure-Centered Architecture** The architecture is designed like a bus, which the components use to interact with each other. Variability is achieved using scripts which defines the behavior of the bus. Binding is done at compile time or in some cases as late as run time. [Sva03]

**Variant Component Specializations** The interface between the components is implemented in separate classes. The correct interfacing class is chosen using a CM-tool, meaning that binding is done at checkout. [Sva03]
Optional Component Specializations  Again, like the previous technique, but there is also a null-component available making the functionalities in the variant component optional.  [Sva03]

Runtime Variant Component Specializations  Inheritance and polymorphism is used to implement the different variants. An abstract factory is used to choose which variant to use. Binding is done at run time. [Sva03]

Single source variation  The alternative to variant segregation is to put all the variants in a single piece of code. The code that is specific for the variant implementations are marked and the common code is left as is, making it possible to extract the different variants from the same source code. This eliminates the redundancy and avoids the double maintenance problem likely to occur for variant segregation. Multiple variance is not a problem, since all the dimensions of variability can be realized in the same implementation. However, if there are many variants and/or dimensions of variants, the markings in the code tend to be scattered, obfuscating the code and making it incomprehensible and hard to maintain. Another drawback when using a single source is the loss of traceability in a variants change process. Since the file that is changed contains all variants, it is impossible to tell which variant was changed by simply looking at what file has changed. [Bab86] [Mah95]

Pohl et al. argues that development using this technique needs to be guarded by architecture rules, determining which single source variation constructs that are admissible, what they should be used for and how they should be applied in the source code. [PBvdL05]

Delta representation  Only one of the variants, or an abstract base, is stored and the other variant implementations are represented using this variant and a set of deltas [Bab86]. This is done in systems like EPOS Software Engineering Database and SMDS’ Aide-de-Camp software management system, which works with document fragments and change sets when representing configurations. A variant is specified using a form of algebra, e.g. ”BASE1.1Win = BASE1.1 + WindowsMods” where ”WindowsMods = CS1 + CS2”. [Mah95]

Conditional Compilation  Precompiler directives like #ifdef is used to choose which variant to use, allowing for all variants to be stored in the same file [Mah95] [Bab86] [Sva03]. Binding is done at compile time allowing unused variant code to be excluded in the binaries for better performance [Sva03]. The binding is controlled using a configuration parameter which is evaluated in the precompiler directive. Since this technique uses text processing instead of the concepts of the programming language, it is easy to generate inconsistent variants by passing combinations of configuration parameters that are not supported [Mah95].

Condition on Variable  Like the previous technique, but the variable can be changed at runtime. Unlike conditional compilation, all code is compiled and needs to be delivered [Sva03]. A similar approach is to use some kind of configuration files, which are read into variables.
**Choice of Technique**

As a rule of thumb, the variant segregation techniques are preferable if the vari-
ants have little code in common, and the single source variation techniques are
preferable if most of the code is common. In most systems both techniques
will be used concurrently and intermixed. A risk when many variant realization
techniques are involved in the build of a variant, is that the build specifi-
cation becomes obscure. Another problem for the building and binding of the
variant features is the possible combinations of independent variants. Theoret-
ically there are very many combinations, since each variant implementation for
a variant feature can be combined with all other variant implementations for all
other (independent) variant features. Many of the possible combinations does
not make sense, though, making this a problematic issue. Which of the variant
implementations that are compatible and can be safely combined is often un-
known, and it would be preferable to have mechanisms to prevent configurations
with incompatible variant implementations and to implicitly choose the variant
implementations that a chosen variant depends on. [Mah95]

2.1.4 Managing the Variability

When the variability has been introduced, it needs to be managed. Managing
the variability includes populating the variability feature with new variant im-
plementations, removing variant implementations that are no longer used and
removing variant features and variation points that are no longer used [Sva03].
A variant management policy can be developed to establish a common way
of working with variants, defining naming conventions for files, branches and
configuration parameters and other guidelines for the management of variants
[Mah95]. Babich proposes a name standard for variants and variant imple-
mentations where the variant feature is used in conjunction with the variant
implementation, creating a intuitive hierarchical name standard [Bab86]. If
branches are used to represent the variant implementations a mechanism to en-
force consistent naming would be preferable [Mah95]. This is not supported by
most version control systems, ClearCase being an exception requiring the user
to choose a predefined branch type.

The build process for variants needs special attention and the steps needed
to build a variant should be documented. Generally used build tools like make
are often insufficient because only the modification times of the binaries are
used to determine if a rebuild is necessary. When a different variant needs to be
built, the parameters to the compiler is changed, but the source files may still
be the same, requiring the binaries to be deleted before a correct build can be
done. Mahler presents a similar tool called Shape, which is specialized in build-
ing variants using a simple and flexible notion of build parameters describing
dependencies and variant binding. Unlike make the tool uses a derivation key
to determine if a rebuild is needed. [Mah95]

2.2 Software Product Lines

A common way of organizing the variability between products in product fami-
lies is to use software product lines. The product line allows for common features
to be shared between products using variability realization techniques [Sva03].
In order to be successful using a software product line, the organization need to plan and be proactive [Bos00].

It is important to realize that the Software Product Line concept can be applied to more than just the source code of the software. All of the artifacts that are used to construct the products, from requirements to test cases, can be incorporated in the Software Product Line approach, making the creation of new products very efficient by reusing as much as possible. However, without proper planning the costs for reuse may be higher than for developing the artifacts from scratch, which were experienced in reuse projects from the 1990s. [PBvdL05]

Using a product line approach can result in benefits considered from different viewpoints of an organization. From an organizational point of view some of them are: decreased time-to-market, better keep up with unpredicted growth, improve the quality of the product [FvDL07], achieve reuse goals [CN02a]. The more complicated the system, the more the cost can be reduced if the performance, distribution and reliability issues are solved in a satisfying way. The product line approach also result in individual benefits such as avoiding painful time-wasting integration phases, fewer stressful delays and will liberate more time to explore new technology. It also increases the possibility to move in a flexible way within the organization, since the knowledge in the organization is more widely deployed [CN02a].

The factors that should be considered when deciding on whether to start a product line are according to Bosch the amount of commonality between the products, the possibility for requirements negotiation and the ownership of the software assets. If these aspects are met to a sufficient degree, a software product line approach is likely to succeed.

The overall goal when developing a software product line, is to optimize the creation of new products which are a part of the product line, in terms of reducing development effort and time to market, as much as possible [Bos00].

In the rest of the thesis, the common and variable artifacts that constitute the base of the Software Product Line will be referred to as part of the platform. The artifacts that reuse the platform or is specific to certain implementations will be referred to as part of the product.

2.2.1 Initiating a software product line

When an organization is to initiate a software product line there are some different approaches available. Depending on if the product line already exists as a family of products or if a new product line is created one can choose to either evolve the product line from the existing set of products or to develop a new product line from scratch. Obviously there are different risks associated with the different approaches. The revolutionary approach of developing a complete new product line has a far higher risk level than to evolve the product line from the existing products. However, with a full product line architecture the reuse of functionality will be grater, the costs will be lower and new products can be rapidly developed reusing functionality from the components of the product line. [Bos00]

When the organization has decided to develop a product line and which approach to use, the first step is to determine the scope of products and features. In this activity the products that are suitable for a product-line are selected. This may not include all products, since some products may have con-
flicts between features that makes them unsuitable for the same product-line. The second step is to determine which of the features in the selected products that are candidates for the product-line. By identifying the features available in each product and creating a matrix showing the products in one dimension and the features in one dimension, the candidate features can be identified as those present in most products. If features overlap in a way that makes them incompatible, Bosch suggests an activity called \textit{harmonization} where the features are split into smaller pieces in order to avoid overlapping in the incompatibility. As described in section 2.1.1, a \textit{feature graph} can be used to understand the dependencies and conflicts between the features.\[^{[Bos00]}\]

It is important to use an appropriate level of abstraction when defining the variability based on the features. The abstraction of the variability needs to match the abstractions used in the domain. If the domain is unknown, incorrect abstractions are made which are not perceived by the stakeholders, causing wrong decisions when the variability is used. Because these abstractions are used throughout the development process, they are costly to repair afterwards. Domain knowledge and product experience is therefore important for successfully defining the abstractions of the features and the variability. \[^{[PBvdL05]}\]

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
Feature/Product & P1 & P2 & P3 & \ldots \\
\hline
F1 & X & & & \\
\hline
F2 & & X & X & \\
\hline
F3 & X & X & & \\
\hline
F4 & & X & X & \\
\hline
\ldots & & & & \\
\hline
\end{tabular}
\caption{Feature/Product matrix for selection of features for the Software Product Line platform}
\end{table}

The number of products that are required for a feature to be beneficial to include in the product-line depends on if the feature set is still manageable. Features only used in one product may or may not be candidates for the product-line depending on the foreseeable use of the feature in future products. If a maximalist approach is used, all features are included in the product-line, and features are excluded when new products are instantiated. The opposite is to implement a platform containing only those features that are used in all products. Of course, the most common approach is somewhere in between. \[^{[Bos00]}\]

One approach to develop the product line is to originate in a product that is used as a test bed for the platform. The reusable parts of the product that are suitable for the platform is then integrated in the platform, allowing for other products to reuse the features. This can be seen as a maximalist approach. \[^{[PBvdL05]}\]

Deelstra et al. divides the scope of reuse into four levels. At the \textit{Standardized infrastructure}-level, only the way products are built is reused. This can be aspects like operating system, database system and development tools. The \textit{Platform}-level denotes that a set of artifacts that capture the commonalities
between the products are reused. When the reuse is extended to functionality that is shared by a sufficiently large subset of products the Software Product Line-level is reached. At this level variation points are added to support the differentiation. The highest level of reuse is called Configurable product family and means that all common and different characteristics is captured in the same configurable product family. This means that products are constructed from a subset of the artifacts and that most development effort is focused on the product line development rather than product development. Product derivation is often automated and applications are created from a configuration specification. This level can also be seen as a maximalist approach, as mentioned above. The effort allocation for product line engineering and product engineering for the different approaches can be seen in figure 2.3. [DSB05]

Figure 2.3: Effort allocation to product and product line development for different levels of reuse. [DSB05]

The product-line architecture and variability realization techniques (see 2.1.3) needs to be decided upon. The design of the architecture is complex because it should support both easy creation of new products and at the same time not conflict with the product specific requirements. If the architecture turns out to be weak once it has been developed, it is generally very costly to address. [Bos00]

The need for flexibility and variability in the product line needs to be addressed, as the costs associated with this is considerable. The trend towards increasing product individualization yields a higher amount of variants and requires more flexibility to differentiate the products. The standardization capabilities offered by a product line approach needs to be used to constrain the flexibility to a proper level, where the costs of flexibility can be balanced with the need for product differentiation. [PBvdL05]

2.2.2 Development process

Pohl et al. proposes a development process where the product line and the variants are divided into separate processes. Both processes address areas like requirements, design, implementation, test and documentation. The provide platform process focuses on the generic aspects and does not realize any products, whereas the provide product process focuses on realizing products using the components and other reusable items created in the product line process. This is visualized in figure 2.4. [PBvdL05]
Figure 2.4: The development process is separated into two processes, where one provide the platform and one provide the product. [PBvdL05]

The provide platform process is responsible for the creation of a reusable platform, where commonality and variability of the product line is defined and realized. The focus is on building a robust platform which offers the capabilities needed by the provide product process. It encompass all types of artifacts from requirements to design, implementation and tests, focusing on providing the appropriate amount of variability in each of these. [PBvdL05]

The provide product process reuses the platform artifacts and are mostly concerned with binding the variability provided by the platform to derive the products from the product line platform. The impacts of the differences between the product and the provided platform needs to be estimated with regards to all types of artifacts. [PBvdL05]

To be effective, the two processes needs to interact in a manner that is beneficial for both. This is simplified by defining the processes using the same sub-processes: requirements engineering, design, implementation and test. The provide platform process is naturally larger in terms of effort, because all variants needs to be considered, while the provide product process uses simpler versions of the sub-processes. The provide platform process also includes a product management sub-process which defines the roadmap and the defined product scope for the platform. The output of the different processes needs to be kept separate, as the output of the provide product process is product specific and the output of the provide platform process is reusable and variable. [PBvdL05]
2.2.3 Variability representation

The variability used throughout the process is in the proposal from Pohl et al. separated from the traditional development process. It is called an Orthogonal Variability Model and spans the whole process from requirements to implementation and test. It specifies where the variability is needed by the use of traceability links to the artifacts. The model is represented using UML where dependencies and exclusiveness between variability features can be specified. [PBvdL05]

There are many motivations for separating the variability from the traditional development process. Variability is recurring in different artifacts and in different stages. By separating the variability model from these artifacts the information can be kept consistent. It also allows for traceability between different types of artifacts, making it easier to determine the influence between artifacts (e.g., requirements and implementation) that is a result of variability. Further, the software development models of today are already complex and if variability is added it becomes overloaded and hard to grasp. [PBvdL05]

2.2.4 Requirements engineering

Requirement engineering is the process of defining functionality required from the stakeholders of a system, and it’s limitations concerning development and maintenance. This involves capturing, structuring, and accurately representing these requirements so that they can be correctly covered. The goal is to ensure these system requirements to be relevant, complete and consistent.

Requirement engineering implies that the requirement techniques and processes should be systematic and modeled in a repeatable way. To decide exactly what to build is the hardest part when building a software system.

The requirements are formulated at an early stage when developing a new system and can describe things as e.g., user-level properties, system constraints, general system properties, how calculations and presentation of how data should be carried out [KS98]. Often requirements are divided into functional and non-functional requirements.

When performing requirements engineering the need of defining a process is of high importance. Such a process must be tailored to the specific organization and should include activities like e.g., elicitation, requirements analysis, negotiation followed by decision making and validation. This work should also result in a process description including the different activities and their structures, responsibility for the activities, activity input-output and the tools being used to support the requirements engineering. The requirement document is where e.g., things like domain information, desired functions, properties and constraints of the system are described. This document is the definitive specification of the system requirements and will from now on be referred to as SRS (Software Requirements Specification). [KS98]

The Requirements Process

When looking on the requirements engineering process at an high level, most of them can be described according to figure 2.5. In the beginning there is always an elicitation set off activity where requirements are formulated through discussions with stakeholders, document studies, possessed domain knowledge and
market studies. Normally, these requirements are analyzed at a more detailed level and negotiations among stakeholders take place. This phase is necessary to establish flexibility and to solve conflicts between the different stakeholders. When this is done, the resulting requirements need to be documented in a clear way so that it is understandable to all stakeholders. To avoid inconsistency between requirements and make sure they are complete, the requirements finally have to be carefully reviewed and validated.

These comprehensive steps in a requirements engineering process are often carried out in an iterative way, so that requirements are continuously being refined, created and removed if needed (see figure 2.5). Simultaneously to this process, requirements management needs to be done which is concerned with managing and keeping track of changes to requirements and handling of the emerge of new requirements.

Figure 2.5: Iterative model for the comprehensive activities in a typical requirements engineering process.

In a requirements engineering process, there are different stakeholders involved, with and without technical knowledge. These can be: System Developers, End-users, domain experts and external regulators making sure the system meets the legal requirements. These people comes from different departments within the company and have a lot of other tasks to carry out and therefore does not always give priority to the requirements engineering process.

Requirements for the Software Product Line

Thorough requirements engineering is a major prerequisite for software product line engineering. This needs to compromise the identification of the commonalities and the variability. Pohl et al. mentions that the CHAOS report (Standish Group 1995) showed that inadequate requirements engineering was a major cause for problems in software projects, making it even more important for software product lines. [PBvdL05]

For a product line, requirements are created and managed for the common and variable functionality within the product line platform (defined in section 2.2). Here, the product line scope need to cover all requirements needed to ensure the variability to be able to create desired products at a later stage. Some of the requirements within a product line will be common to all products, and some of them will vary. The analysis of the requirements to identify those that are common and those that are variable or product specific is a very important part of the requirements engineering practice from a software product line perspective. It is therefore preferred to cover which variation points to be
supported, since these regulate the variation within the product line. [FvDL07] [PBvdL05]

It is important to realize that the requirements engineering sub-process in the provide platform process does not generate a requirement specification of a particular product, but the common and variable requirements for all foreseeable products in the product line [PBvdL05]. The requirements are used as an input to the architectural design. Regarding to the software architecture and the provide platform process the non-functional requirements specifications are of high relevance [Bos00]. This concerns both operational (performance, robustness etc.) and development (reusability, maintainability etc.) quality requirements.

To be able to facilitate the use of the great amount of requirements used in a product line, several requirements, related to each other can be deployed into features, as mentioned earlier in section 2.1.1 and 2.2.1. These features then represent a logical unit of functionality. Sometimes, product requirements within such a feature are overlapping with requirements in other features in the product line (i.e. one requirement is included in two features). Then, some kind of dependency relation between these features needs to be introduced [Bos00].

In the proposal by Pohl et al. these dependencies and constraints are documented in the Orthogonal Variability Model, separated from the traditional process. In the requirements engineering sub-process this variability model of the product line is defined as an abstraction of the variability with traceability links to the requirements, possibly via features. It defines what will be variable (variant features), where the variability should be introduced (variation points), what variants are available (variant implementations for population) and the dependencies and constraints that has to be considered when the products are derived by binding the variability. Because the model is intended to be orthogonal and used by all sub-processes, the definition of the variability in this model will be consistent for all artifacts [PBvdL05].

Often, requirements are inherited from a previous product line to a new one. In this case, when new functionality is developed, it is of high importance that these are assigned to the requirements in the product line. [CN02a]

There are some potential problems associated with requirements engineering. The main risks constitute of not being able to keep requirements up to date, wrong documentation. Inappropriate requirements can also result in wrong variation points [CN02a] which will reduce the flexibility in the current product line.

Requirements for the Products

In section 2.2.1 it is described how the product line scope was determined by using a feature/product-matrix. Since this is probably done by the requirements engineers of the provide platform process, they may not fully understand the product needs that are known by the requirements engineers of the provide product process. There may also be a considerable amount of time between the definition of the product line platform and the definition of the products, which causes differences between the platform capabilities and the product requirements that are now known [Bos00].

The requirement elicitation for the products mostly consists of deriving requirements from the platform requirements, rather than defining new require-
ments. However, the delta between the platform requirements and the proposed product must be analyzed. Either the requirements can be reused by binding the variability available in the platform, or the requirements introduces new functionality which must be implemented. There may be product specific requirements (e.g. from a customer) that has not been captured in the requirements engineering of the provide platform process. These requirements needs to be elicited and compared to the platform requirements to evaluate the effort needed. If this is done early, trade off decisions can be made to reduce the effort by increased reuse. [PBvdL05]

This delta analysis can also be performed using the feature representation of the requirements. The product features are compared with the platform features and the dependencies and constraints between the features are considered. During this activity several problems can occur [Bos00]:

- The product feature can overlap an existing platform feature. The product feature should then be redefined to only include the requirements that are not fulfilled by the platform feature.

- A platform feature is removed in a product, but there are other features that depend on the removed feature.

- The product feature conflicts with a platform feature. In some cases this can be solved by making the platform feature a variant feature and introducing a new variant implementation for the product feature. However, some conflicts can affect the very structure of the platform and the use of the software product line for realizing the product may not be justifiable.

The product requirements that are reused should be documented in the software requirement specification with traceability links to the platform requirements. In the model proposed by Pohl et al the Application Variability Model is also part of this specification, stating the selected variability bindings of the platform. If the delta analysis discover that new variability is needed for the product, the variability model deltas should also be specified in this document. [PBvdL05]

2.2.5 Configuration Management

Software Configuration Management is defined as the discipline of organizing, controlling and managing the evolution of software systems. This is done in order to minimize the mistakes and eliminate confusion, in other words, to increase the productivity. The more people tied to a software project, the harder it becomes to carry out the project in an efficient way because of more time spent on communication. The reason why this happens is often due to the lack of conventions and rules of good conduct missing, i.e. coordination failure [CN02a].

In Configuration Management (CM), there are three common scenarios illustrating why problems occur [Bab86]. The double maintenance problem, i.e. when several copies of the same software is maintained. The shared data problem where new changes of a particular software can interfere with the progress of others, and the simultaneous update problem which actualize the risk of new work being overwritten by others. Hence, CM is not just an issue for auditors,
CMs and people involved with quality assurance issues. To be able to avoid
the above mentioned problems even members of the developing teams need to
master the disciplines of CM. Obviously, to be able to introduce them to the
CM principles, they need to be motivated by becoming aware of the potential
benefits. It is also important to have well defined CM practices in place at an
early stage.

**Product line aspects**

In the context of product line development, CM takes on a special significance
since there must be a configuration maintained for each variant. Therefore we
will end up with several configurations, partly for the core assets, partly for
each product, but also for the entire product line. To be able to manage these
configurations a qualified CM tool, supporting e.g. branch and join, is recom-
mended to be used. It is also important that the CM process offers an easy way
to set up an initial configuration for a new variant and that the CM keeps track
of the versions of the tools being used (e.g. compilers, make tools). Managing
the configuration of the different products constitute an important part of the
work for the CM department. This makes the management more complex and
demands keeping track of that components developed by one development group
can be used in parallel by other groups.[CN02a]

To ease the CM work, most often, some kind of plan is established. This
plan normally contains principles for change control, organization roles and
definitions of the life cycle of the product. There are several standards currently
being used (such as IEEE and ANSI) that helps organizations to attain a good,
excogitated CM plan. CM is also used as a basis for release baselines and helps
organizations to track change requests.[CN02a]

It is important that the CM process is formed to ease the setup of the initial
configuration for a new product. This makes the access to detailed specifica-
tions describing which components to be shared within the Product Line crucial
[FvDL07].

**System models and views**

In a Software Product Line not only the implementations of the variant features
can vary, but also the composition of the system, the part list, is subject to
variation. The components included in a variant configuration can vary, and
also the structural relationship and dependencies between them. This is often
specified using either a system model or a view. The system model specifies
which components from the product line that are used in the configuration,
which versions of the components, and how they are related. The view concept
is more abstract, working with rules to select versions instead of specifying the
versions directly. These system models are often stored in files, allowing the use
of the variation techniques described earlier to create new variants. [Mah95]

**Product derivation**

Deelstra et al. have identified a two-phase process for product derivation, i.e.
the initial and iterative phase. [DSB05]

In the Initial phase a first configuration is created from the product line
assets. This can be done either by assembling products from a subset of the
shared artifacts or by selecting a closest matching existing configuration. In the later approach the selected configuration is often a product developed in the latest project, since it contains the most recent bug-fixes and functionality. Another aspect is that repeat customers are likely to desire new functionality on top of the functionality they have ordered before. When a first product configuration is ready, the initial phase is ended with a validation. In this activity the extent of adherence to the requirements is determined.

Until the product is finished, the initial configuration is modified in the Iterative phase. This can involve reselection of components or resetting parameters, resulting in a new product configuration that is validated. When the product configuration sufficiently implements the requirements the iterative phase is ended and the product is ready.

When the organization has reached the level of reuse described by Deelstra et al. as Configurable product family, the components are often stabilized. The architecture is enforced because no product can or needs to deviate from the commonalities and differences specified in the product line architecture. Because of this the return on investment for formalizing the product derivation process is substantially higher than at the reuse level for a software product line. [DSB05]

2.2.6 System architecture

The software system architecture is represented by a set of decisions made early in the design phase and can be viewed as a bridge between the requirements and the implementation phase. These decisions are the most difficult to change later in the development life cycle and therefore becomes crucial to get right [CN02a]. It represents an overview of the system and allows designers to reason around whether the requirements could be met. At this level outlines can take form containing e.g. data flows, data transformation and where input-output streams should be handled. Here, the designers are given opportunities to estimate buffering capacities, computation costs, set up boundaries and determine possible risks [Gar00]. In other words, the system architecture describes the overview structure and is supposed to improve the communication between stakeholders. Since this design moment is so critical, it must be evaluated continuously and at an early stage. There are several advantages of evaluating the architecture already when being in the developing phase because different aspects can be weighed in before the decisions are made. In the beginning, the hardware aspects and performance factors are often unknown. Therefore, restrictions regarding performance should be included when the architecture is established [CN02a].

Furthermore, it is important to separate the components and the functionality that allows them to interact. Considering evolution aspects such a separation makes it easier to change connection mechanisms. This result in a more flexible design and facilitate reuse possibilities [Gar00].

Product line aspects

In a product line approach, not only requirements for the products should be taken to account, but also requirements for the whole family of systems. Software architecture therefore engage two different roles in a product line approach.
One architecture for a product line as a whole and one for each product. The requirements for the whole system include relationships between the requirements for the different products.

These two roles are normally evaluated in different ways [CN02a]. The product line architecture should be evaluated regarding to robustness and to secure that it can work as a basis for future products. The evaluation of the particular products has more to do with meeting the specific behavior requirements.

It is in the architecture design of the provide product process that the variability is bound and the product is derived from the platform. This is done by selection and configuration of variant implementations for each variant feature. Product specific adaptations needs to be evaluated regarding implementation effort and impact. If structural changes requires an effort similar to the development of the product from scratch, the requirements may be rejected. [PBvdL05]

2.2.7 Software Evolution

Evolution can imply either the changes or creation of new requirements necessary to fit in to the architecture, or it can imply changes of the software because of maintenance. In many cases, new requirements can not be supported by the current architecture which then must be able to be changed. If there is no stable process for this, it can lead to problems with mismatch between documents and break the design of the Software, even by series of small changes. [FvDL07]

In order to able to maintain a "healthy" software system that is likely to survive evolution, it becomes important to foresee what type of changes that can occur [Sva03]. It is therefore necessary for the architecture to support these kind of changes. However, there is a high risk in trying to support to much flexibility which makes the software system to complex to handle. Therefore, constraining the amount of variability possibilities is crucial. In reality, it is more realistic to identify general types of evolution than creating a rigorous plan, predicting and specifying exactly how the software is going to evolve.

According to Svalnberg [Sva03], there are two major problems when trying to support the evolution of a software product line. The first one concerns the difficulties in decreasing the amount of changes needed for support evolution, and the second one concerns the complexity necessary to manage evolution, and how it could be minimized.

Evolution is highly connected to software reuse, both reuse over time and when components are reused over a set of products. By reusing software between products the costs can be decreased and more time could be spent on improving already existing components [Sva03].

2.3 Other case studies

2.3.1 Celsius Tech Systems

In the 90’s, a case study was made by the Software Engineering Institute (SEI) on a Swedish defense contractor (Celsius Tech Systems) that had succeeded to introduce a product line approach. The need for introducing a product line occurred when Celsius Tech needed to build two larger systems in parallel. At this time they discovered the potential benefits and opportunities in selling
series of systems related to each other, rather than offer a few separated specific systems.

Early, Celsius Tech saw the need for introducing flexible sets of blocks to be able to generate new systems in an easier way. On the basis of this, a decision was made to create a new approach which would comprise both new hardware, new software and a new way of handling the development. A product line was introduced consisting of an architecture dividing functionality into families of application components. From here, a product were supposed to be put together by selecting components from these component families. The new way of working highly increased the reuse of the existing software. This was managed by applying a strict configuration management to the components and consistently treat the components as product line assets, separated from the individual systems or products. Some components were developed to meet the different needs from different customers. These components were then always tested to determine whether they could be used for product line reuse in other areas in the future.

The architecture was separated into layers divided into general base functionality (e.g. operating systems) at the lowest level to more detailed functionality (e.g. customer specific) on the top level. Here the critical interfaces between the layers needed to be well defined. Before introducing the product line they had their functionality divided into packages. When this division was made Celsius Tech realized that the granularity of the packages, in which the software was grouped into, were too detailed and the amount of them to great. Therefore, it was necessary to decrease the number of packages by making them larger to be able to handle them in a more efficient way when configuring the systems. To be able to manage these packages they were also distributed into system function groups that were divided into functional areas. This made the division of the functionality more clear and easier to overview.

To be able to make the new way of working consistent over time, Celsius Tech put significantly high resources on educating the employees in the beginning. This was done to make sure the employees really could understand the concept and to communicate a strong commitment to the new product line approach.

The main cause for their success with their product line was the architecture and it’s division into layers. If the architecture was not stable the components would probably not constitute a stable system and therefore it became important to focus on the architecture early in the process. Other areas that were contributing to the success of introducing the product line was the new definition of the software package granularity and the clear and proper education of the employees. As a result of their introduction of a product line approach, their hardware to software cost ratio changed from having been 35:65 into a 80:20 relation. [LB96]

2.3.2 Mobile browsers at Nokia

At Nokia, a Software Product Line was used to develop mobile browser products, both for their own handsets and as a software product for other customers. [Jaa02]

The requirements for the product line was originally written in specifications for each product release and did not specify requirements for the whole product line. Later, they tried to allocate the requirements to product line components,
but the customers and managers did not have the technical experience needed and would rather discuss features and products. Requirements were therefore mapped to product releases and the allocation to components was done later in the design phase. This was done by having the engineering teams analyzing the specifications and extract the generic functionality into the product line core. The goal was to make the core as big as possible to maximize reuse.

The implementation of the product line was separated into one core team concentrating on the common parts of the software and some other teams handling the product and customer specific implementations. An architect in the core team created a functional specification document which documented the product line architecture including the variability and information on how different products should be developed from the reusable components.

Development artifacts like documents, test cases and code was managed in a centralized configuration management system. The article does however not mention how the commonalities and product specifics was stored in this system.

Test cases was developed based on the functional specification and was shared between products. They invested a lot in the implementation of reusable test suites, which became an important part of the product line’s core assets. The testing was not simply done on the core asset components, but the functionality was rather tested in the context of each product.

To minimize costs, functionality that had potential for reuse but would not be directly reused was not incorporated in the core assets of the product line. This was avoided because of significantly delayed product releases due to delayed core asset creation. Instead, the products was built first and then the components was extracted into the product line. This made the architecture of the product line less coherent because it was assembled from components originally intended for different architectures.

By using a product line approach, Nokia increased efficiency through reuse, accelerated product implementation, lowered costs and increased quality. However, the product line required extra resources in process and tools development, core development and product management.

2.3.3 Bosch Gasoline Systems

Robert Bosch GmbH is one of the largest automotive suppliers. Within the division for gasoline systems more than 1000 developers are creating software for engine controls. Basically each new engine control yields a new software variant and with thousands of variants there is a good basis for introducing a product line. [FvDL07]

The development was already based on a powerful platform and the product line engineering offered a good framework for improving the quality and reducing the costs. The product line was developed around a new architecture, but assets were derived from the existing asset base. The existing components were evolutionary redesigned to fit the product line architecture.

New tools and data formats were required to support the development and maintenance of the product line and its developed artifacts. This includes tools for feature modeling, architecture documentation, interface documentation, linking between feature model and implementation and feature-based product derivation. Due to the lack of commercial tool support for these tasks, Bosch had to develop their own tools. However, the prototype versions of the tools...
were not user friendly and in most cases required more effort to use than they could save. The lack of tools lead to a delay of one to two years for the process introduction and is therefore seen as a major risk for achieving the intended benefits of a software product line.

2.3.4 Cummins Engine Inc

Cummins Engine Inc. is one of the worlds largest manufacturer within the diesel engine industry, providing the basis for thousands of separate engine products. In the middle of the 90s, the company was facing a huge challenge. They had six ongoing software projects, and another twelve planned in the near future. At this time, each group was using its own processes, its own architecture and sometimes even its own language, and the lack of plans and documentation was evident. One of the managers for the technical organization then realized that these planned projects could never be pulled off in time if continue working like that. This made him pause all ongoing projects and instead allocate resources from the main project to start building generalized core assets that could be used by all the products, both planned and ongoing. At the same time, a standard software process was introduced and a dedicated development team was formed to be able to establish common development and configuration management and provide tools needed for this.

In the beginning they choosed one of the legacy systems, being under development with sufficient documentation and stability, as the starting point for the product line core assets. They analyzed the requirements in the domain to be able to establish the requirements for the product line. Then, all features were identified and categorized according to being unique or common and finally prioritized. The common features were then implemented by the core asset group, and the specific features were distributed to be implemented by the different product teams.

Products in this domain involves a great amount of variability. Thousands of parameters are available to control the engine applications and the software is using thousands of #ifdefs to select functionality to be used in the different builds. For this a great effort was put to cover this in the requirements and in the architecture.

In Cummin’s case, the introduction of a product line was crucial to be able to deliver the planned products and the experience from the product line introduction was positive. The planned projects were carried out on time, and they estimate that what used to require 350 engineers, today could be done by 100 software engineers working in the new product line approach. Cummins were able to build over 1000 of different products or variants based on their new product line with higher quality. Many of the employees were complaining in the beginning and some of them even left. This despite that the company undertook much communication effort to make the employees aware of the potential benefits of the new way of working. However, those who stayed finally saw the light in the end of the tunnel. [CN02a]

2.3.5 Ricoh Image Memory Handler

Ricoh is one of the leading developer and producer of office equipment in the world, including copiers, printers and other office equipment. One of the main
issues in Ricoh’s products is how to process image data with small memory space but still be able to offer a good performance. This case study is focusing on their component Image Memory Handler (IMH) who’s techniques are used in many of Ricoh’s products.

In IMH, conditional compile is used to enclose differentiated parts of the code with `#ifdefs`. Investigations made showed that only a small amount of these were used for actually make differentiations. Instead, many of them were used for debug reasons or by developers to guard their own code. This was seen as a problem since large use of conditional compile sooner or later becomes unmanageable for most people to understand and to configure. They also discovered great amounts of duplicated code that were similar or even identical, mainly as a result of code being copied. Except for this their investigation also showed shortcomings regarding reusability, maintainability and lack of updated documentation.

When attend to these problems Ricoh tried to improve the reusability without changing existing functionality. They began refactoring by modifying preprocessor statements, renaming file and functions and improve the design by introducing subcomponents. Since almost all source code files were importing the majority of the header files, these were solved by splitting common data types and data definitions into separate files in order to avoid importing parts not being used. Finally, all duplicated code were weed out manually by implementing external functions for code commonly used by different parts. [RKY06]

2.3.6 Philips Consumer Electronics

Philips Consumer Electronics is one of the largest consumer electronic companies in the world, having 16 000 employees all over the world. The televisions are responsible for one-third of the turnover, and the software product line set up for the software in the televisions was studied in the case study. [FvDL07]

The change to a product line required a new organization were product-oriented development was turned into a single development organization including a platform team and product teams. The architecture for the product line was developed with a compositional approach using a component model with implementation in C and interfaces in Interface Description Language (IDL). The use of IDL allows for reuse of both components and interfaces, to create alternative implementations of an interface. The product line consists of a composition graph of components, which is the high level variability mechanism. Low level variability is achieved using parameters organized in special interfaces. The components are grouped into a hierarchical structure of private and public packages.

The introduction of the product line and the associated organizational and architectural changes took Philips three years to implement. Because of the long introduction time without any intermediate results, the introduction has failed in other parts of the company. A more incremental approach is therefore welcomed, but Philips also has experiences from architecture changes that were too slow to obtain significant results within a few years.

One failure factor that has been identified is an improper balance between platform and product engineering, giving problems with the generic parts of the platform resulting in that no product could be released from the product line. An other problem has been too many dependencies between components, resulting in much time spent on integration problems when modules with many
dependents are changed.

2.3.7 Siemens Medical Solutions

Siemens Medical Solutions develops software for different types of hospital applications, from patient registration to software for X-Rays, MRI- and CT-scanners. [FvDL07]

The product line architecture is based on a client-server architecture. The variability is achieved by choosing different components for the client in the detailed design phase. However, some of the variability is bound at runtime.

The focus of the case study is on the introduction of a new testing strategy where variability defined during requirements engineering is maintained throughout the domain engineering until testing. During this introduction some observations were made regarding variability. The textual descriptions of the variability were interpreted differently by different stakeholders, such as product management, software architects and programmers. The variability needs to be validated and verified between a customer and a developer, which was achieved by the use of explicit variability modeling instead of textual descriptions. The model led to discussions which finally led to a common view on variability among all stakeholders.

2.3.8 Salion Inc.

Salion Inc. is a small company developing Revenue Acquisition Management solutions tailored to the unique needs of automotive suppliers. The system helps suppliers to manage opportunities, analyze requests, support decisions and collaborate with its customers and sub-suppliers. [CN02b]

The software product line was developed using a reactive approach, meaning that one product was first developed and this was then used as a platform which is extended and configured to create new products. The products differ in many ways, mostly in the customer’s work flow, set of input screens and set of sub-products that are chosen.

The variation points are strictly on the level of files and folders. To manage the variation points a tool called Gears is used, which identifies artifacts that is common and variable and uses this information with a standard build tool to build different variants. Because the tool works on the file level, it can be used with different types of artifacts such as requirements or test cases. The tool is based on the reactive way of working where the product line is seen as one large product that is adapted to form new products, rather than as a large set of core assets.

The success of Salion’s product line is in the case study explained by its unified vision, process discipline and its focus on developing one system rather than many variants. The last reason was also emphasized when the company deviated from its strategy and tried to develop a generic form system which failed to support any of the product variants, instead of creating a system that supported one of the products and tailor it for the other variants.
2.3.9 Thales nederland B.V.

Thales Nederland B.v. develops ground based and naval systems in the defense area. The case study focuses on the software for the Naval Combat Systems family. The system's main purpose is to integrate all weapons and sensors on naval vessels, handling Command and Control and Combat Execution, both for real world and simulated worlds. [DSB05]

The asset base was developed in the 1990s and include a virtual machine, a publish/subscribe mechanism and a real time database. The common parts involve real time storage, message transportation, virtual world creation and dynamic reallocation for applications. The asset base is fairly stable and consists of approximately 1.5 MLOC. It can be classified as a platform, but it does not reach the level of reuse denoted as a software product line, although the company plans to extend the scope of reuse.

An average product built on top of the platform contains about 37 components and several MLOC. There are several thousand lines of parameters that needs to be set to configure the platform for a product. Because of this, the product derivation process starts with selecting the most similar product from the most recent project. The configuration is modified to comply with specification for the new product, by reselecting components and resetting parameters. The architecture is not changed directly, but implicitly by the reselection of components. When the initial configuration is done the system is installed in it’s target environment and the system is validated. If the configuration does not pass the validation, the configuration is changed and it is revalidated. The changes are handled by product CCBs and if needed by platform CCB.

The documentation of the platform has deviated from the actual functionality as the platform has evolved. The product derivation process therefore strongly depends on knowledge possessed by the employees rather than documented formal descriptions.
Chapter 3

Case Study

3.1 Introduction

Sony Ericsson Mobile Communications (SEMC) is one of the leading companies within the mobile phone industry providing mobile multimedia devices including feature rich phones, accessories and PC cards. SEMC produce products containing applications for mobile imaging, music playing, communications and entertainment and uses the common communication opportunities of today through 2G and 3G platforms. The company was founded in 2001 by Ericsson and Sony.

Today SEMC has several thousands of employees distributed in different sites over the world e.g. in USA, China, Sweden, Japan etc.

Since the number of mobile phone models have increased lately and are being released in many different variants, there is a good basis for running Software Product Lines. Each of the product lines cover different technologies and markets. There is e.g. one low-end, one mid-range and one high-end product line. The mid-range product line is based on an in-house developed software platform supporting a range of different phone types in the segment. This is the product line that is evaluated in this thesis. The high-end product line is based on Symbian OS offering smart phone products.

Though the product lines are quite separated in the implementation and testing phases, they to some extent have the same requirements and therefore share the same requirement base and requirements management system.

The product lines lets SEMC produce many phone variants with short lead times by using the same software assets specially configured for each produced product. The configuration is done both for differentiation purposes and for hardware specifics that vary between products. Differentiation can be in the form of advanced feature sets and branding that is specific to a product or product type. Most of the configuration is however dependent on hardware characteristics of the products, like available hardware features including e.g. WLAN and GPS but also variants of common hardware like number of screens, screen resolution and camera properties.

In this case study the processes for achieving variability between the products is investigated. This is done from the requirements engineering through the implementation phase to the testing phase, focusing on how the variability is represented, implemented, specified and eventually bound during the product
3.2 Evaluation of current process for variability management

In this chapter an overview of the different processes at Sony Ericsson Mobile Communication (SEMC) involving variability and configuration of products is presented.

The content of this chapter is a result of internal document studies and interviews made with different roles in the development process (see section 1.2.3). As mentioned, these interviews were performed in a semi-structured way and the people were selected both randomly and from recommendations from other people in the company. After each interview, a summary of the meeting was written on the basis of the notes taken by both of the students. This was done to minimize misunderstandings.

The different perspectives presented in this chapter describes how the stakeholders perceive that products are created.

3.2.1 Software Product Line

The product line architecture at SEMC is organized in clusters in two dimensions (see figure 3.1), one for product segments or product technologies and one for time or evolution. Each product segment has it’s own platform and code base which evolves through time. In each of the clusters there is one main product built from the platform representing most of the platform features. In a product line manner this product is scaled down to create sub products and new variants for other markets and customers. Few of the sub products originating from the main product contains new features, but it sometimes happens.

The use of a main product for the Software Product Line is similar to the approach described by Pohl et al. where one product is used as a test bed for the platform and the product features which are suitable also for other products are incorporated in the platform [PBvdL05]. This is also similar to the Configurable Product Family described by Deelstra [DSB05].

In the case study from Salion Inc (section 2.3.8) one product was first developed and this was then used as a platform which is extended and configured to create new products. The product line is seen as one large product that is adapted to form new products, rather than a large set of core assets. This is similar to the SEMC product line approach.

3.2.2 Overview of the current development process

The development process is organized in two kinds of sub processes where one supplies the platform and the other creates a product. For every cluster there is one platform project, one main product project and several sub product projects, where the platform project uses the full development process and the product projects uses a scaled down version (figure 3.2). This separation of the platform and the products into different processes has many similarities with the process described by Pohl et al. earlier [PBvdL05].
Figure 3.1: SEMC Software Product Line organization. There is one product line for each level of technology. Each product line is divided into several clusters representing the evolution. In each cluster there is one main product and many down scaled sub products.

In the following chapter a discussion on how the actual process is carried out based on different stakeholders perspectives is presented. This is ended with a model of the defined development process.

### 3.2.3 Organization

Some of the roles and organizational groups in the development process is especially interesting for this evaluation. Each technical area of the domain has it’s own group of requirements engineers responsible for covering the progress in the focused field. They need to maintain a list of features for the roadmap needed to keep SEMC in the forefront of the industry. Their involvement in the projects are mainly focused on the platform project where they supply high level requirements and prioritization of features that are candidates for the platform.

The development teams which are working close to the requirements engineers are responsible for implementing the software in the platform. They are involved in the platform project at an early stage to estimate the effort needed for the features and to review the requirements. Each feature is assigned to a primary development team which is responsible for the implementation of the feature in the module.

Before the modules are delivered a platform verification team tests the module.

The different modules needs to be integrated and compiled to a full system. This is done by the Product Configuration Management team which manage the different variants and versions of the products created from the platform.

The compiled system is tested by a product focused testing organization, Product Software Verification.
3.2.4 The requirement management process

The process for requirement management at SEMC is currently focusing on software requirements, and there are several areas affecting it. In this case aspects mostly from customers, standards, legal issues and suppliers need to be taken into account. The products must be differentiated and customers must be able to differentiate the same product.

At SEMC, the requirements engineers are responsible for the technology roadmap, which serve as a source to the requirements in the platform project. This roadmap includes requirements known from product planning, associated areas, previous clusters and customers.

Changes and Configuration Packages

The SEMC requirement database contains tens of thousands of requirements. In the past, product differentiation has been specified for each and every requirement, imposing a great risk for mistakes. To be able to manage these requirements in a better way, two types of containers are used to bundle requirements for different purposes, Changes and Configuration Packages.

Changes are describing which new functionality that shall be implemented in a cluster of a platform. A set of requirements are grouped into a change, containing new functionality or improvements. The changes are used for scoping and prioritization of the cluster. Because of this, legacy requirements are not associated with a change, since they are already implemented in the platform. For each cluster in the product line, there is a document stating the scope of the cluster by listing the changes. Using the link between requirements and changes, this document can be used to compile a requirement specification for the cluster which can be used to implement the platform. This scoping document does not state any form of variability and is rather focused on the development of the platform.

The requirements are also grouped into Configuration Packages used to differentiate the products by picking different packages for different products. A Configuration Package is a set of requirements grouped to form a logical unit of functionality, as described in [Bos00]. Every requirement has to be associated with one or more Configuration Packages. The Configuration Packages should be of a size that is relevant for configuring the product from the product
planners perspective. Example of such could be "FM Radio Basic" and "FM Radio Advanced". This will be discussed further under "Configuration Package Modules".

Basically Changes are used when scoping the platform, and Configuration Packages when configuring products. In figure 3.3 it is illustrated that Configuration Packages also changes from one cluster to another. In the beginning CP1 only consists of Req 1. In the next cluster CP1 is extended with a change C1 as a result of a new requirement Req 2. This means that Configuration Packages with the same name can vary due to new requirements in the different clusters.

![Figure 3.3: Configuration Packages and Changes](image)

Adding/Changing requirements

Sometimes, one or several requirements need to be changed. But a description of a requirement that have been used in other contexts can not be changed since that will result in inconsistency with previous products. Therefore, a requirement that is already in use, is not allowed to be updated. Instead it has to be replaced by a new requirement.

Configuration Package Modules (CPM)

The requirements engineers lists the Changes and Configuration Packages (CP) for their area of expertise in the Configuration Package Module (CPM). A CP could be seen as a module, which together with other modules in the platform can be used to create a product. Naturally these modules have dependencies between each other, as described by Bosch [Bos00], and some of them are mutually exclusive. For example, "FM Radio Advanced" might have dependencies to "FM Radio Basic", requiring the later CP to be chosen if the first CP is used (see figure 3.4). There may be two different variants of a music player, which are equal and mutually exclusive, meaning that a product cannot have both CPs. These dependencies are also documented in the CPM.

Configuration Packages that are common for all products in a cluster are marked with an attribute stating that these packages cannot be removed from a product configuration.

By having this information separated into Configuration Packages but still connected to each requirement through links, the model is very similar to the
Orthogonal Variability Model proposed by Pohl et al [PBvdL05]. With the capabilities of defining dependencies and constraints between Configuration Packages and the traceability links to the requirements, the resemblances are remarkable. The major difference is that the Configuration Package model is not yet used outside of the requirement management process. The resemblance is further emphasized in section 3.2.5 where these are connected to the product process.

Configuration Dependencies

Variability between products are not only due to differentiation of the software. There are also dependencies to e.g. hardware which makes individual requirements valid or invalid for different products. This is specified by the use of Configuration Dependencies on the requirement level. There are four categories of configuration dependencies: form factor, hardware, market and customer. Using these, a requirements engineer can specify that a certain requirement is or is not valid for a specific product.

Refinement of requirements

When the platform requirements has been written, associated with Configuration Packages and scoped they are refined and detailed by the developers. However, the requirement process described is currently being implemented and has only reached the requirements engineers at this time. This means that the concept of Configuration Packages and Configuration Dependencies does not reach the developers in the refinement step of the process. The ambition of SEMC is that this information should be delivered to the developers when the process is fully implemented.

Today the connection back to the requirements is insufficient, and changes to functionality are often not reflected in the requirements. This means that some of the requirements becomes out of date when functionality changes, and
therefore becomes obsolete. However, according to the defined process the requirements engineers are responsible for updating requirements due to feedback from the developing groups. Here insufficiency in the communication have been discovered.

Requirements Engineer’s Perspective

The requirements engineers are the only group who has started to use the new requirements process to it’s full extent. They use the Configuration Packages and Configuration Dependencies, at least for new requirements. The platform perspective of the requirements engineers takes the focus off the products and therefore also to some extent off the variability. The use of Configuration Packages and Configuration Dependencies increase the focus on variability while still preserving the platform perspective.

Figure 3.5 shows how the requirements engineers perceives the process of how new products are created. The requirements engineers are visualized as the oval marked “Req”. Their input consists of the roadmap, product concepts and customer requirements. They produce the Configuration Packages and high level requirements for their specific area, focusing on the platform. They are also the main responsible for the scoping of the platform with the use of the earlier described changes. They experience that their output is used and refined by the developers, but are aware that the concept of Configuration Packages and Configuration Dependencies are not used by the developers yet. They also recognize that the Configuration Packages should eventually be used by the product planners for easier definition of new products.

Figure 3.5: Requirements engineer’s perspective on the creation of new products.

When asked about how the variability was achieved before the use of Configuration Packages, the requirements engineers have a hard time to answer the question. Their platform focus does not leave room for variability and they are not concerned with what features that should vary between products. Some larger differences between product families are considered and the variability is stated in the requirements texts, but most variability is implicit and the developer needs to recognize this. The requirements engineers also suspects that the
variability is achieved later in the provide product process through communication between the product planners and the developers.

Customer requirements should go through the requirements engineers, but the requirements engineers are aware that late customer requirements are often communicated directly to the developer via bug reports instead. They see this as a problem which to some extent makes the requirements and the software inconsistent. According to Clements, it is of high importance that new added functionality is clearly connected to the requirements [CN02a].

**Requirements Coordinator’s Perspective**

The requirements coordinator works between the requirements engineers and the developers to communicate the requirements so the developers can refine them. The requirements coordinators rarely experience that variability is communicated through the requirements. Some requirements coordinators have gotten information about customer specifics informally on requirements presentations from the requirements engineers. This is then documented in the refined requirements. As mentioned the Configuration Packages and Configuration Dependencies are not copied to the developers’ requirements document yet, which yields the informal communication of this information. Some of the requirements coordinators cannot commit to the Configuration Packages proposed for their area, because the packages are not synchronized with the variability of the platform implementation, which delays the process.

The requirements coordinators share the common opinion that the developers receive most of the configuration instructions through bug reports issued by product projects, the customer responsible and the testers. Sometimes the product project directly contacts the developers asking for help with configuration. They also experience that the configuration to some extent is carried out by the Product-CM team. The perspective is modeled in figure 3.6.

![Figure 3.6: Requirements coordinator’s perspective on the creation of new products.](image)
Product Requirements Coordinator’s Perspective

The Product Requirements Coordinator (PRC) is responsible for the communication of the requirements between the product planners and the requirements engineers, on a product specific level. The coordination is focused on transferring the conceptual and less technical requirements from the product planners into more technical specifications. In the new requirements process this will typically be specified in the Product Configuration Specification, which is described in section 3.2.5. At this time the product specifics are rather fragmented in different documents. The product requirements coordinator contribute to the creation of the Product Impact Analysis and the Product Comparison Chart, which will be described later. Some of this information originate from the Concept definition (section 3.2.5). Except for this, information is collected manually to a high extent, often by performing detective work and by contacting people with the right knowledge.

The product requirements coordinator’s perspective of how new products are created is modeled in figure 3.7. It is more detailed, but the PRC believes that the configuration is carried out by the developers using product specific requirements specifications. Later the PRC mentions that these kinds of specifications has not been generated for a while, making the actual configuration process unknown.

![Figure 3.7: Product Requirements Coordinator’s perspective on the creation of new products.](image)

3.2.5 Product planning & Product Project

Product planners are responsible for defining new and attractive products from the platform available in the cluster. This is mainly done in two deliverables for each product, the Concept definition and the Product Configuration Specification. This part of the process is concerned with creating the product rather than providing the platform.
Concept definition (PRB)

The product planners establish a concept document intending to describe the product considering commercial overview, price range, competitor analysis and gives an overview of the top level requirements of the product. The document is intended to work as a guideline until the requirements specification is established and is intended to lie as a basis for the Product Configuration Specification which specifies the product from the platform.

The concept definition is in its first version a vision of what the product planner would like to be included in the product. In this version it is used as input to the requirements engineers in the process of defining new features, where the gap or delta between the product concept and the available platforms requirements is analyzed. This is similar to the delta analysis described earlier.

The features are scoped for the platform and based on this scoping the second version of the concept definition is updated to reflect what the product will actually contain.

Product Configuration Specification (PCS)

The Product Configuration Specification specifies the configuration of a product concerning both software and hardware. The document should be the main specification of how the product is composed from the platform. When this thesis is written, this deliverable is part of a new requirements process which is introduced at SEMC and is therefore not used yet.

In this deliverable the product is specified using the Configuration Packages defined in the CPMs, Configuration Dependencies which apply to the product and a list of hardware components. The idea is to make this document the main specification for what is and is not included in a specific product. The document is realized as a module in the requirements management system which should be filled in by the product planners with help from the product requirements coordinator. It should be used as input to designers and testers on what is valid for a specific product.

Again, the model is similar to the Orthogonal Variability Model proposed by Pohl et al [PBvdL05] by separating the variability from the rest of the development process. The Product Configuration Specification correspond to the Application Variability Model which in the proposal is part of the Orthogonal Variability Model.

Product Impact Analysis

The product projects are using a Product Impact Analysis document which is used as a plan for the work needed for the specific product. Beside special customization and file system of the product, this document also contains a section where configuration deviation from a baseline product is specified. This section contains some of the information which will typically be in the Product Configuration Specification when the new requirements process is implemented.

Product Comparison Chart

To be able to easily compare different products and get information on what features and properties that apply for specific products, a Product Comparison
Chart is written for each cluster. This document is unofficial and not part of the process, but is created by the Product Planners and the Product Requirement Coordinator by detective work. Information is gathered from a variety of documents and verbally from different sources. It contains much more detailed information about the deviation between products than the PCS, but the information can be derived from the PCS. The document is widely used by both developers and testers to get an overview of the products features. The SEMC ambition is to compile this document from the PCS in the future, making it a valid and correct document.

Product Planners Perspective

Product planners belong to the marketing section in the company and are often limited concerning technical knowledge. Therefore their task is rather to create an attractive product [FvDL07] than to perform the configuration of it.

The product planners perspective of the creation of new products is modeled in figure 3.8. At an early stage they write the concept definition of the product which is used by the requirements engineers in the creation of the platform. Later in the process they are involved in the creation and configuration of new products by creating CCB issues regarding both new functionality for specific products and for the platform and also regarding configuration of existing functionality. The CCB issue is, if accepted, sent to the developer team responsible who then performs the implementation or configuration. The differentiation achieved in this manner is not documented in any kind of product specification, but only in the CCB minutes of meeting. Their perspective is that this is the way that the products are configured and that the actual configuration is carried out by the developers.

The concept of Configuration Packages are only briefly known by the product planners. They are not aware of when or how it is supposed to be used, but are positive to the idea.

The product planners have not experienced the need for more detailed documentation of a products functionality than what is delivered in the concept definition.

Figure 3.8: Product Planner’s perspective on the creation of new products.
3.2.6 Software design process

The design of new features is documented in the Feature Impact Agreement and the Software Impact Analysis. These documents are written by the development groups and are based on the refined requirements. There are also Software Architects who works across the development groups to ensure that the overall architecture is "healthy".

Software Impact Analysis

The Software Impact Analysis describes how new requirement impact the existing platform software. The requirements are analyzed and distributed over the architecture, and possible implementation strategies are suggested. Risks are analyzed and efforts are estimated. The document also specifies which Feature Impact Agreements that are needed for cross group development.

Feature Impact Agreement

When different groups of developers needs to communicate during the development of a feature, a Feature Impact Agreement document is created to keep track of what work is needed from the different groups and works as a contract between the groups. This document may also be established for larger features developed by a single team of developers.

The document is connected to variability by including a chapter which states the planned configurability of the functionality. However, it is used whenever teams need to communicate and this chapter is therefore seldom used. Another drawback is that this document is generally not updated after the design-phase and configurability created during the implementation is therefore not documented here.

3.2.7 System variability architecture

The SEMC platform offers some different variability techniques with different binding time and purposes, basically by the use of static configuration and dynamic configuration. Static configuration is applied at compile time and generates one compiled and linked software for each combination of chosen variation points. Dynamic configuration is used to control the behavior of the software in a product or variant of a product. This configuration is checked at runtime and does not affect the compiled software. Here, values are added to different customization sets or to different versions of the file system. Examples on things that can be configured dynamically could be either customer or market specific details concerning themes, ringtones, startup themes etc. This configuration is normally carried out when the product is started for the very first time.

Today, variability is realized statically by importing software modules in the product configuration file. These modules points out the actual source code to be used when building the final product (see figure 3.9). All these software modules have their own configuration file where settings concerning the functionality of the module can be configured using the configuration at product level. These files uses its own syntax defined by SEMC.
External binding

The variation points of the SEMC platform that are realized with external binding are bound at compile time using conditional compilation. There are typically two techniques used for this type of binding, which are covered in the following sections. This thesis will focus on the external binding in both the evaluation and the improvement proposal. The internal binding is also of high interest to the subject, but is out of scope for this thesis and will therefore only be described briefly.

Variant segregation

The software is organized in different modules or components which can be used to build different variants. The selection of modules and module versions is done in a cluster-specific file using an in-house developed configuration interface. This file contains definitions of the available variants that can be built and depending on which variant that is chosen different modules are included. This technique is previously referred to as Variant Architecture Component and Optional Architecture Component. The configuration interface uses Conditional Compile syntax to create the resulting makefile.

Single source variation

The same conditional compilation architecture that is used to specify which modules to include can also be used to accomplish single source variation in the source code. Each module can define variables in the interface that is needed to configure the different variations inside the module. These variables are then set in the configuration interface for the cluster and used in the source code to include different code fragments. This technique is very flexible, offering capabilities to conditionally include files, which is the preferable way of separating variant feature implementations, or to conditionally include single lines of code, which is a more flexible but less maintainable way of implementing the variability.
Variation points

The conditional compile variation points has a naming standard for keeping them understandable. The syntax includes a prefix that is common for all variation points, a feature name which should reflect what feature the variation point affects and optionally an attribute for the configurations that is not of on/off-type.

Internal binding

The technique previously referred to as Condition on Variable is used to create customer specific variants with late internal binding. By specifying settings in an XML-file which is put in the products file system and read at the first startup, variation can be achieved late in the development process to comply with customer requirements.

When a new variation point for internal binding is created, the developer needs to consult a specific customization team which is responsible for handling the variation point and the population of this.

Another use of the Condition on Variable technique is the users settings in the product. This is also a form of variation since the user can choose to turn features on or off at runtime. The different modules needs to cope with this variability technique in the communication and dependency on other modules.

Ambitions in variability architecture

In the guidelines for software configuration SEMC emphasizes the importance of decreasing the conditional compilation in the modules to minimize problems with compile time dependencies. This should be done using interfacing techniques such as IDL and CHAPI, which increases the modularization and verticalization of the software. A thorough description of these techniques is beyond the scope of this thesis, but the techniques offers an interface for dynamical linking between modules making the dependencies between them "softer".

Each product is currently maintained in many different software variants for different customers. The Software Architects strive to replace the conditional compile architecture associated with customer specifics with dynamic configuration using the XML-interface, to reduce the number of software variants that are needed to be built for each product. The goal is to have only one software variant per product.

3.2.8 Implementation

In the implementation phase the developers use the refined requirements and the Feature Impact Agreements. Unfortunately, the work done by the requirements engineers concerning variability does not reach the developer in a formal way.

Developers perspective

The developers perspective of how new products are created is modeled in figure 3.10. Normally, the developers in the different development groups never
sees requirements concerning variability. Either they get extractions from requirement management tools containing requirements without variability information, or they get their input through meetings and other informal communications. Things concerning variability are most often communicated to developers via team leaders as change requests at a later stage. These request mostly comes from product projects and testers who often are unable to figure out whether an error occur because of bad configuration or errors in the source code. This is determined by the developers. However, a very small part of the change requests from the testers concerns incorrect configuration. The change requests from the product projects are often what is used to configure the products. Sometimes they contact the developer directly regarding configuration of a product. The developers opinion is however that the Product-CM does most of the configuration, at least for the sub products.

The creation of new variation points is done in the provide platform project and is therefore often based on assumptions made by the developers out of previous experiences and informal communication with involved people. If developers are uncertain of what value should be assigned to a variation point he or she normally asks someone about it. The main opinion is that this information is rather floating and tied to specific individuals and is hard to find in any formal documents or specifications. A widely spread experience is also that there are room for improvement in the way variation points are documented. Because the lack of concrete information about the variability, sometimes communication between development groups and product planners takes place to solve those issues.

When variation points needs to be set or created, developers forward their work to their internal Configuration Manager who then forward the changes of the configuration file to the Product Configuration Managers. The Configuration Manager normally receive this code via email or through other informal documents.

Figure 3.10: Developers perspective on the creation of new products.
Requirement communication

Despite that the requirements engineers have grouped the requirements in to Configuration Packages and specified the Configuration Dependencies, this information is at this stage not copied to the specification that is communicated to the developers. Instead requirements are communicated to them through scaled down extractions from the requirement management system and through informal communication from change requests or verbal directives from the team leader. This means that this part of the requirement process is passed over which makes the developing teams unsure about what the product specific requirements looks like. Therefore, when interviewing them, it is hard for them to explain exactly where the information about the variability descend from.

Motivations for new variation points

During the interviews the stakeholders (see figure 1.1) were asked about the creation of new variation points. From these interviews it has been seen that the creation of new variation points is done by the software developers during the implementation of the software platform. There are a variety of reasons for creating these variation points:

Assumptions Since the focus at this time is on creating the platform, there are generally no specifications of what functionality should be variable between products. The products has not been specified yet, so the developers cannot determine this from comparison between product specifications. It is also common that customers have specific requirements on the functionality, but this information is not available at the time of platform implementation. Because of this, developers creates variation points in a preventive way based on assumptions of what functionality that needs to be variable. This makes it easier and less time consuming to make changes based on customer requirements. Unfortunately this result in many variation points never being changed, and therefore becomes needless.

Ease of creation A conditional compilation variation point is created by editing two files, the source code and the configuration file, which the developer has easy access to. Late requirements from customers requires rapid implementations of the variability. If internal binding is to be used, another team needs to be consulted and the variation point needs to be approved, which clearly is a longer process than creating an external binding using conditional compile. Because of this developers tend to solve things using the latter technique.

Version Control Sometimes late products in a cluster uses more functionality than previous products. When the platform evolve inside a cluster, variation points are created for the new functionality in order to be able to remove the functionality for maintenance releases of previous products, instead of using branches. The motivation for this is that bug fixes which apply to the whole cluster would have to be delivered to both branches, which requires more effort from the developers.

Integration When different groups of developers have dependencies between their modules, a variation point is created to remove the functionality
when the groups deliver their work to integration. When all the participating groups have delivered their work, the functionality is turned on. The variation point is never used again, but will still be present in the source code and the configuration file. This is done to avoid simultaneous delivery of all participating groups.

**Debug** The architecture of the system modules requires that variation points that is shared between modules are declared in the configuration file. Because of this, a development group spanning many modules often declare variation points for debugging purposes in the configuration. These variation points are never used for the configuration of products, but will still be present in the configuration file.

**License costs** Some parts of the software are related to license costs, such as e.g. codecs for audio and video. It is therefore desirable to be able to exclude these parts when not being used since the extra cost for this will be unmotivated.

**Customer specifics** The largest customers have special variants of each product which contains customer specific functionality. This functionality is to some extent controlled using conditional compilation, though the ambition at SEMC is to only manage one variant and use internal binding for customer specifics. As mentioned above, conditional compilation is easier to achieve than internal binding, which counteracts the vision of one variant.

**Product specifics** Sometimes the requirements states that the feature should be configurable, which makes it obvious that a variation point is needed. Other times hardware properties like form factor yields different implementations of the requirement, resulting in the developer creating a new variation point with an associated configuration parameter. The latter scenario means that the new parameter could be set by looking at the hardware configuration, but instead the parameter is published to the product configuration interface which creates a redundancy which may result in confusion and maintenance problems. The choice to configure hardware dependency on product-level instead of hardware-level is based on tradition.

### 3.2.9 Product Configuration management

The Product-CM teams are responsible for integrating, building and managing the variants in the clusters. This partly involves setting the configuration parameters in the software to create the main product in the cluster. In order to do this in a convenient way, the traceability from configuration parameters to the requirements stating which variant feature to use is vital. This part of the process is subject for improvements, which are a part of the research questions for this thesis (section 1.2.1).

**Customization vs. Static Configuration**

*Static configuration* is applied at compile time and is a method being used to control what parts of a software that should be included in a specific product
in a platform cluster. In SEMC’s case, this can be e.g. market or customer specific versions of the software. Static configuration generates different software for each combination of variation points. In contrast to static configuration, Customization implies dynamic configuration which offers a way to control the behavior of a product or variant applied at run time. This way of configuration does not affect the compiled software.

Primarily the architects vision is that static configuration should only be used for debug purpose and to switch out code affecting license costs and should not be used for configuring interface availability. In other words static configuration should only be used to configure products, not features. Principally, customization should be used to create variants for different markets and customers etc. i.e. change the behavior of features.

The vision is to only have one static configuration for each product and avoid over design the configuration. However, this is not the situation today.

Configuration file

When setting up a new product, a configuration file is used. Today this file contains all the configuration parameters deciding what functionality should be included and which values to be set in different areas, both concerning hardware and software. The file is used when making a new build of a product. The configuration parameters are available either directly in the file, or indirectly by being imported from other configuration sources. Each developing team has its own section in the file, where they can choose to switch the functionality. The larger this file gets, the more problematic it becomes to handle since approximately 20 different developing teams are making changes to it simultaneously. There is also a great variety between high level to more detailed low level configuration parameters. Especially the low level points are hard to handle since there are so many of them. Also, many of these configuration parameters have not been changed in a long time and it is therefore complex for CM to keep track of them all and knowing what values to assign them to.

As mentioned, the configuration parameters declared in the configuration file lack proper documentation, and there is no clear mapping between their names and any requirements. This makes the configuration file difficult to clean up, and therefore obsolete configuration parameters are still present in the file.

Besides the configuration parameters, the configuration file also specifies which version of each module to use. The file corresponds to the System Model described earlier by Mahler [Mah95].

Merging and compiling

Because all configuration is kept in a single file which is updated by every development group, the Product-CM needs to merge this file for every development team delivery. Most of the times this should not result in merge conflicts because the development groups have their own section of the file. However, the file also contains a list of the module versions which is most often changed by the development groups. These changes are unwanted by Product-CM and therefore cause somewhat false merge conflicts in the file, generating extra manual work for the Product-CM.
Cluster Configuration Specification

When configuring a new product, CM need to know specific information about e.g. camera chip, flash lights and other hardware included in the product. This information is mainly derived from a hardware specification containing information about the hardware setup for each product in a cluster.

Configuring new product lines

This section briefly describes the procedure at the Product-CM department of how the main product in a cluster is created.

When a main product in a cluster is created from the platform this is done by copying the configuration file from a similar previous product. Then, some configuration changes are made, mostly regarding to the hardware such as choice of Camera chip and memory type etc based on the specification described in the previous section. Since the amount of configuration parameters in the configuration file have increased, and are vaguely documented they have become unmanageable for the Configuration Managers to keep track of. This forces them to leave some of the configuration out and mainly rely on the configuration made in previous products. This is in contrary to VanDerLinden et al [FvDL07] where the easiness of setting up the initial configuration is strongly emphasized. The way of working is similar to what Deelstra et al described as the initial phase for product derivation [DSB05].

When a new product have been set up, it is built and sent to the Product Testers. They test the product trying to discover software errors and functionality that might be missing. At this stage it is often difficult for them to determine whether errors depends on faulty configuration or software errors. Therefore they create a bug report toward the development groups with a description of what is not working. It is then up to the developers to determine what actually caused the problem. The errors are corrected by the development groups and new source code is later sent back to the CM Department. They are then responsible for merging the delivered code from all the development groups. The way of working is similar to what Deelstra et al described as the iterative phase for product derivation [DSB05]. Figure 3.11 shows an overview of how the main product is created.

The process has many similarities with the process described in the case study from Thales nederland B.V. (section 2.3.9), where a similar product is also copied and the configuration is achieved in an iterative process.

Configuring new products

This section briefly describes the creation procedure of a sub product.

When a sub product is created it is first copied from a previous similar product in a similar manner as for the main product. Though, the CMs responsible for the sub products tries to configure the whole product, both regarding to hardware and software. This is done by gathering information from all possible sources and specifications. Especially by using a document specifying the product structure. The last-mentioned mostly contain information about what hardware configurations and language settings to be included in the product. Since there is no document in place that covers all the configuration of a prod-
uct, much information is informal and duplicated. This is what Babich calls the double maintenance problem [Bab86].

If it is unclear which value should be assigned to a specific variation point it is often set by assumption or by comparing with configuration made in other products. From CM’s perspective, lack of documentation and the unclear mapping between the specifications and the configuration parameters makes it even harder to set these values. This leads to time consuming investigations and an unsureness in the configuration of the product. An indexing tool exists and is used to find the variation points associated with a configuration parameter, and thereby determine what they do.

The CMs responsible for the sub products are not mainly focusing on merging the implementation from the different developing teams. This work is mainly performed by the CMs responsible for the main product.

Normally it takes a week to carry out the initial configuration of a sub product and the correctness of the configuration is vaguely verified.

The perspective of the sub Product-CM is modeled in figure 3.12.

Figure 3.11: Main Product-CM’s perspective on the creation of new products.

Figure 3.12: Sub Product-CM’s perspective on the creation of new products.
Configuration metrics

In order to understand how the configuration is changed over time some metrics have been calculated. The configuration file was picked out for each label of the code base of each cluster. The differences between each configuration file was calculated to get metrics on how many parameters that was added, deleted or changed. These data are visualized in figures 3.13 and 3.14. Note that over 60% of the configuration changes are made after the software has been shipped to the testers (MS Alpha). This supports what have been seen earlier in the interviews, where the developers describe that they configure the products based on bug reports and change requests.

When this was written the configuration had over one thousand different parameters available at product level, spread across a configuration file of thousands lines. These parameters then control over 30 thousand variation points in the source code with different levels of granularity (e.g. file inclusion and line inclusion). This means that each configuration parameter controls an average of 28 variation points, which suggests that most of the variability is quite fragmented. The source consists of millions of lines of code in more than ten thousand files, giving an average of 250 lines of code per variation point.

![Changes over milestones](image)

Figure 3.13: Changes to the configuration over milestones
3.2.10 Product test

The product testers are responsible for securing the products. Their work is carried out in different test cycles e.g. build check, sanity check, release test and system test. Even though tests on the platform already have been performed, tests on the functionality needs to be done again since it is impossible to cover all configuration combinations when testing the platform [FvDL07]. However, most effort is put on testing new functionality. A lot of extra focus is also put on key functionality in the product and exploratory testing where areas which are specially known to contain problems are tested. In these areas no test specifications are used and the test engineers are allowed to be more creative in their way of working.

Test cases are normally inherited from previous products in a similar manner as functionality is reused in the different platform clusters. These tests should normally be verified but due to lack of time this is sometimes neglected. In the early stages not all things are tested since there might be functionality that have not yet been delivered.

The team testing customer specific functionality is involved in the last 10-15% of the testing phase. The late customer requirements and bug reports are mostly regarding customization and configuration and does not contain new functionality. It is therefore not interesting to involve the requirements engineers in this phase. The testers send bug reports to the development teams who corrects the configuration. In most cases there are already variation points available to correct the configuration, but sometimes new variation points needs to be created. Since this is late in the process the creation of the variability point must be quick, which may result in the developer choosing external binding.
3.2.11 Documented software engineering process

The defined process, which may not always be the same as the actual process, is modeled with focus on variability and configuration in figure 3.15. This model is based on SEMC documentation. The question mark denotes an undefined responsibility for configuring the products, which could not be found in the defined process. Which specifications that should be used for configuration is also undefined in the process, though there are documents that are good candidates and have been described earlier.

Figure 3.15: Model of development process with focus on variability and configuration. The requirements from the different stakeholders are transformed into platform requirements, a scope and a number of Configuration Packages. The developers refine the requirements and create design documents, which are then used to produce the actual software. In the Provide Product process, the product projects create a product concept, which is used as a basis for the Product Impact Analysis and the PCS, which should be used to generate the Product Software Requirements Specification. How and by who the products are derived from the platform using these documents is not defined. It is rather implicit that the P-SRS should be used by the developers to create the products.
3.2.12 Holistic view of process

Most of the stakeholders, including the requirements engineers, requirements coordinators, product planners and the developers, have a common view on how the products are created (figure 3.16). The Product Requirements Coordinator has the most deviant view (section 3.7) which is more detailed and corresponds to the documented process (figure 3.15). Because the process does not explicitly define who is responsible for the product configuration activity, it is not possible to relate this to the stakeholders’ common view.

Figure 3.16: Holistic view on how products are created.

3.2.13 Evolution

The evolution of SEMC’s product line is facilitated by the division into different clusters containing different products. These clusters overlap in time to some degree, but the code base is reused in each cluster. In the requirements engineering process the requirements and features are scoped for the cluster in which it will be implemented. Requirements that has been scoped and implemented in earlier clusters are part of the product line and will be part of products in the following clusters. If features are to be removed from the product line new requirements needs to be written to support the removal of the feature from the product line.

One entity that has been referred to many times in this case study is the Configuration Packages used to handle the variability at the requirements level. The evolution of a Configuration Package follows the clusters (see figure 3.3), meaning that the same Configuration Package will contain different features and requirements for different clusters, a fact that needs to be considered when products are specified using these entities.

3.3 Problem situation

This section will focus on the problems that the different roles are experiencing in their work connected to variability.
3.3.1 Requirement Engineers perspective

The requirements engineers are concerned with the platform and are rarely involved in the different product projects. They map their requirements to Configuration Packages and specify Configuration Dependencies where applicable, but are not concerned with how these are used to form and differentiate products. Some of the requirements engineers did not know that this information was not propagated to the developers, which all requirements engineers sees as a problem that needs to be corrected in the implementation of the new process.

The requirements engineers are responsible for the creation and management of the Configuration Packages. They emphasize that these needs to be reviewed and discussed with the developers to reflect the variability possibilities of the platform.

The requirements engineers should receive customer requirements and possibly scope them into the platform. However, customer specific requirements that arrive late are not received by the requirements engineers, but rather sent directly to the developer as a bug report. This is seen as a problem by the requirements engineers, because it invalidates the platform requirements.

3.3.2 Requirement Coordinator perspective

The requirement coordinators are responsible for communicating the requirements to the developers for further refinement. A problem today is that the new requirement process is only implemented for the requirements engineers, and is not yet used by the developers. This means that Configuration Packages and Configuration Dependencies does not reach the developers and the information is to a high extent unused.

The requirement coordinators can not accept Configuration Packages that does not comply with the possible variability in the platform today, which slows the implementation of the new process. The requirements may be implemented but the variability is not in line with the specified Configuration Packages. This is because requirements engineers have created the Configuration Packages without collaboration with the development teams.

The main problem for the requirement coordinators is that there is no complete specification of what should be included in a product or what should be variable in the platform. This means that there is no way to inform the developers what variability that is needed, resulting in the preventive excessive configuration.

3.3.3 Software Architect perspective

The main concern for the architects is to keep the overall design of the platform organized and "healthy". With regard to configuration the main problems are the inflexible dependencies between modules and the compilation time.

Many of the existing configuration parameters in the software cannot be changed due to complex dependencies from other modules. If the configuration parameter is changed and the variant feature is removed, the platform will no longer compile. This means that the configuration parameter is no longer variable and should be considered obsolete.
The intentions are to solve the dependency problems by using a dynamically linked framework based on IDL and CHAPI, where components can interface loosely by asking each other for the needed functionality. This allows for variability by simply removing the binary of a module without having to recompile the software. Hardware dependencies should be realized by a special module which offers the same type of flexible interface. Lower level variability should be achieved with internal binding at runtime.

The utopia is that all external binding is replaced by this dynamic variability, giving a more flexible platform with reduced build times because of less dependencies. This is far ahead and a sub target is therefore to move all customer specifics to internal binding, meaning that only one variant for each product needs to be compiled and managed.

3.3.4 Product Planner perspective

When Product Planners wish to bring late features in to the platform they contact the Change Control Board which contact the developing teams asking for an estimation of the effort implementing it. Based on this estimation, a decision is made and if approved, the Change Control Board give clearance to the concerned developing team to implement the feature.

In the process of planning new products there often occur different opinions between the involved groups. On the one hand, the Product Planners possess a clear vision of how the product should be shaped, both regarding to functionality and form factor, to fit into the market strategy of the company. On the other hand, designers and engineers are rejecting parts of the concept since they better can determine what costs are tied to realizing the proposed functionality, regarding to time and resources. This most often results in Product Planners have to give up some of their requirements. Approximately, a majority of the requirements delivered from the Product Planner are not supported by the platform, which often is experienced as frustrating. To be able to increase the possibility to carry through their position during the meetings with the Change Control Boards, Product Planners sometimes contact developers to increase the insight and to get their acceptance for their vision. It can also be the case, that functionality proposed by the Product Planners are approved, but later descoped because the concerned developing teams are experiencing resource bottlenecks.

The Product Planners spends time on writing CCB requests for their products. Because there is no complete main specification for a product these decisions are not documented properly. From the Product Planners viewpoint, there have not been made any notices about the lack of product specifications among developers and testers. Though, they are aware that responsibilities regarding to the ownership of the configuration of new products are unclear, they see them self as potential owners of documents like Product Configuration Specification in the future.

3.3.5 Product Requirement Coordinator perspective

The Product Requirement Coordinator spend a lot of time putting together the Product Comparison Chart which specifies the differences between products in the platform. Since this document is used by many different roles within the
company, the Product Requirement Coordinator is to some extent experiencing pressure to deliver this. If this information instead is available through the Product Configuration Specification, the detectives work done by him can be avoided and much time can be spared. As mentioned earlier, this document is not official and therefore never reviewed or approved. The updates of this document are not properly managed. This is a critical issue, which may put the final product at risk to be different from what was intended from the beginning.

3.3.6 Developer perspective

In general developers are not experiencing any major problems with the way the configurability is handled or managed today. Mostly they prefer solving issues by introducing variation points over using customization which takes more time and involve other teams. Though, the intentions from the Software Architects point of view is to introduce dynamic configuration and customization to as high extent as possible. However, this requires extra time and sometimes this is not allowed because of narrow deadlines.

Sometimes developers experience that some of their configuration parameters are not properly delivered from one platform cluster to another. This is most probably due to an early branching of one platform cluster to the next cluster, resulting in that development on the first cluster done after the branching may not be merged to the new cluster, illustrated in figure 3.17. There are no distinct rules on how merging between the different platform clusters are performed which can cause these kind of problems.

![Diagram](image)

Figure 3.17: Why variation points sometimes can disappear.

Developers only contribute to the configuration of sub products when a bug report is received. They often know what to do and does not recognize these bug reports as a problem. Many of the reports does not actually require a change in the software, but are faulty because the testers are expecting functionality that is not supposed to be in the product. The developer then has to correct the tester and explain that the functionality is not available in the product due to e.g. hardware dependencies. The time used by the tester to test functionality that is not available, and the time used by the developer to find the correct
information is due to unclear or fragmented specifications.

There are also change requests originating from Product Planners regarding configuration of a product. These are relatively easy to handle and are not seen as a problem by the developers, but do require time from many of the stakeholders. The changes are rarely documented in the specifications, yielding a divergence between the software and the specifications available.

Pohl et al. mentions that the configuration task should not be completely left out to the developers without restrictions, because this can lead to unwanted product configurations and software performing in an undesirable way. The developer may not find the right configuration within reasonable time because the number of possibilities is too large [PBvdL05]. In the SEMC case the developers are in some degree performing the configuration of the products, and in combination with the high amount of variation points the problems described by Pohl et al. may be applicable.

The unrestrictive rules for creating new variation points results in excessive creation of them, often with unsatisfying motivations which are not associated with product configuration. This leads to a large configuration file which is difficult to overview and manage. As mentioned, the variation points are lacking proper documentation and the knowledge is therefore kept in the minds of the developers. This might make the process immature and vulnerable to employee turnover.

As the software evolve these variation points contributes to the increased complexity. Development groups which have many conditional compile statements have experienced that the source code becomes less readable and hard to manage. At a certain point this yields a clean up and reorganization of the group’s source code. The complexity may impact the product’s quality and the effort needed to implement new functionality.

This complexity due to absence of rules is also described by Pohl et al. who argues that architecture rules are needed to govern when, how and why new variation points can be created. [PBvdL05]

3.3.7 Configuration Manager perspective

The Configuration Managers for the main product are spending much of their time on merging the configuration file delivered by the different development teams. Problems arise when conflicting changes needs to be merged and when the products are to be compiled. The size of the configuration file, the amount of configuration parameters and the granularity of them makes the configuration hard to overview and results in constant simultaneous updates which needs to be merged. Because of the complex dependencies between modules, the integration results in compile errors or faulty builds and the developers needs to be consulted. To find out which developer is responsible can be a time consuming work which is highly desired to be avoided.

The lack of product specifications makes it impossible for the Configuration Managers to configure the product properly. The product concept document and informal communication is used to determine which previously created product that is most similar. This product’s configuration file is copied and some of the hardware specifics are updated according to a hardware specification containing all product specifics for the cluster. Due to the lack of more detailed product
specifications, the main product is not configured further in the opening step. Detailed configuration and software specifics are there left as is.

The Configuration Managers responsible for the sub products performs more configuration on their products. By informal communication and using various sources of incomplete information the product is configured by comparing the configuration settings in previous products. The absence of complete specifications makes a configuration audit of the product difficult.

Even if there would be complete and correct specifications for each product, the complexity of the configuration parameters would make it more difficult to configure the product. The granularity of the configuration parameters is too fine grained and is rather on the level of configuring the features than configuring the products. The complexity of the configuration parameters makes it hard to determine if a feature is available in a product, which must be determined for a configuration audit. This will probably have an impact on the product quality, since it is possible that both testers and developers fail to discover the faulty configurations. Developed functionality may be accidentally left out in the product, resulting in unnecessary development efforts and incomplete products and faulty configuration may give unexpected behavior which may not be detected by the testers.

Another time consuming aspect is the great amount of cycles being iterated before a product is configured satisfyingly (figure 3.18). Here, the overspent time is disseminated over each cycle, and would be desirable to decrease since it is not unusual that the number of cycles exceeds 100 iterations.

![Diagram](image)

Figure 3.18: Iterative configuration of new products.

### 3.3.8 Product Test perspective

When errors are found in the product it is often difficult for testers to determine whether it occurred because of incorrect configuration or not. In some cases the test engineer might think that the functionality have not been delivered yet but actually it is in place but was never turned on in the configuration.

The general opinion among the engineers testing the products is that they are experiencing difficulties in finding specific information concerning variability. Documents being used for this is e.g. the Software Requirement Specification, platform scope document, assignment specifications from customers,
impact analysis made by developers and informal documents describing the differences between the products. Over all, finding detailed information and valid specifications is encountered as a time consuming job and is also often received through informal communication with people possessing the knowledge. The change request system is also used as an information source on how the different products should be configured, which is in line with how the product planners use this system to get the developers to configure their products. This highly increase the probability to not cover all functionality that should have been tested and will most likely affect the efficiency in the way of working and finally become a quality issue for the product as a whole. This also increases the divergence between the available specifications and the software. Today, there is therefore no guarantee that the tests will cover all of the requirements.

The use of the Product Comparison Chart, which is neither reviewed nor approved or managed, can also impact the quality and coverage of the testing phase. Because the configuration activity is not properly managed there will be many unnecessary cycles between developers, CM and test. These cycles results in unnecessary regression testing and testing of software which has not been properly configured. Questions are raised and bug reports are sent to developers who needs to investigate both if the feature should be available in the product and how it should be configured.

3.3.9 Overall problem picture

According to Van Der Linden et al. [FvDL07], the Configuration Manager should be responsible for maintaining the configuration of all variants and make sure the functionality for the whole range of products is covered. In this case, the responsibilities in the configuration process are unclear and the actual variation points have no specific owner. The configuration of new products takes long time and is achieved in an iterative manner between developers, CM and test. Since the configuration is not properly reviewed, there may be functionality that is accidentally missing in the products. This can lead to instable software and a greater need for maintenance work. If the configuration is incorrect, testers may perform testing that is to some extent unnecessary in the early iterations.

The iterative configuration of products is similar to what Deelstra et al. have described. The SEMC process have both the initial and the iterative phase, but there is no formalized knowledge for product derivation. [DSB05]

The lack of an owner for the configuration process also results in an absence of clear, specific and strategic goals and long term improvements.

In the process proposal by Pohl et al. the Application Architect, or Product Architect, is responsible for binding the variation points of the platform to create the specific product [PBvdL05]. Such a role is not present in the current SEMC organization, because most of the development and architectural activities are platform focused. Pohl also discusses the effort needed for the binding activity and argues that this depends on the abstractsions used in the variation points and the traceability between the variation points in the architecture and the variability requirements. In the SEMC case both of these aspects may contribute to an inefficient configuration of the products.

As mentioned, the unrestrictive rules for creating and managing new variation points results in excessive creation of these. A great amount of variation points becomes obsolete either because they were not created for product con-
configuration purposes or because of complex dependencies. It is undefined who is responsible for removing these obsolete variation points which makes the configuration file hard to manage and overview. The granularity of the variation points may be a problem for the appointment of an owner for these. They are used to configure the products and the platform focused development teams would therefore not be suitable as owners. Yet, they are too complex to be owned by the people responsible for the product configuration and differentiation.

Pohl et al. discusses the cost of flexibility that needs to be coped with by standardization of the product line. The introduction of variability that is not used leads to additional costs for developing, managing and also removing the variability. It is also very important to use an appropriate abstraction level for the variability, ideally matching the commonly used domain abstractions. [PBvdL05]

In the SEMC case the flexibility is, in the sense of amount of variation points, too great and offers much more configuration capabilities than is needed for product configuration and differentiation. The variability capabilities needs to be more standardized and less detailed to handle the costs associated with the flexibility.

At Cummins Engine Inc (section 2.3.4) the software was also configured using thousands of conditional compile parameters, resulting in a great effort needed for configuring the products. At Ricoh 2.3.5 most of these parameters were not used, but made the configuration hard to overview and the configuration process unmanageable. In the case study from Celsius Tech Systems (section 2.3.1) the granularity was also too detailed to use when configuring products. This was solved by decreasing the number of packages by making them larger, resulting in a more efficient product derivation process.

Throughout the organization the greatest common challenge is that there are no complete product specifications available. This may lead to the following problems:

- Time consuming detective work where information is gathered through informal communication and unofficial documents.
- Faulty and unnecessary bug reports.
- Double maintenance of the fragmented product information that exists in different documents and versions throughout the organization.
- Faulty configuration.
- Unique and undocumented knowledge possessed by individuals.
- Difficulties in verifying the configuration of a product.

The problems are tackled by the use of unofficial documents specifying the product characteristics for both hardware and software. The document is created in an informal way and is neither reviewed nor approved, but yet used throughout the organization to fill the hole of the incomplete product specifications. The fact that the document may not be reviewed make its content unreliable. Because it is not part of any process, it is maybe not updated regularly as decisions are made. When the information is used by developers and testers this may result in to some extent different products than was originally planned.
3.3.10 Quality and Efficiency

By using the concept of Software Product Lines including reuse of software, many benefits are achieved such as reduced costs, time to market and the quality improvements of the software. This have been shown by many studies [FvDL07].

However, this approach does not guarantee these kind of benefits without a well structured process followed by everyone. In the SEMC case, it has been revealed that the process rules are not always followed and that exhaustive specifications are not yet available. As mentioned earlier, this affect several steps in the product development life-cycle. Especially considering the unclear responsibilities regarding the configuration of new products and that there is insufficient access to information about how a main or sub product should be configured. This means that information about how a product should be configured is floating within the organization and since so many activities lack validation and reviewing there is a potential risk that functionality in the products is inaccurate or missing. This also means that today's way of working is rather inefficient since similar things are done segregated from each other and much information is duplicated.

Obviously, sooner or later when working this way, gaps between the Product Planners intentions and the final product may occur. Hopefully, this can be avoided by making the work across the company more consistent so that the work of product planners, requirements engineers, developers and testers could be more coherent. This will be the focus in the next chapter, where some proposals will be presented for the purpose to improve the current way of dealing with the configuration of new products.

3.4 Software Architects’ proposal

This section describes the configuration model proposed by the software architects at SEMC. This was developed in parallel but independently from our proposal (section 4.1) and was presented in the end of the thesis work. This model is described to show how the principles of our proposal can be applied to different syntaxes and architectures.

The architects’ main goal is to accomplish modularization by making a transition from static configuration into more dynamic configuration. In contrast to our goal, their primary focus is not to ease the way products are configured today. This proposal does not take association to any requirements into account, and is rather designed from an architectural and software developing point of view. This model focuses mostly on long term improvements.

3.4.1 Bundles and Packages

The proposal constitute a concept of Packages and Bundles. A package is defined as a set of software that can only be built in one way, i.e. without variability options. This means that no variation points and no variable build options need to be available. Hence, the static configuration is done with combinations of packages rather than with variation points. This decreases complexity and build time.

Bundles constitutes collection of packages (according to figure 3.19) that can be built separately from other Bundles. In a Bundle, packages can be both
imported and exported. It is also possible for a Package to be shared by several Bundles. It could also be the case that a Bundles contains the same packages as other Bundles but is configured slightly different. All modules are distributed into Bundles which are supposed to be defined and owned by the developing teams.

### 3.4.2 Static configuration

Until the concept of Bundles and Packages are fully supported by the software architecture, the static configuration used today is preserved in legacy configuration files which can be imported into Bundles. Obsolete configuration is moved to a platform configuration file which holds configuration that is the same for all products. Product specific configurations will be moved to product files which are imported in product specific Bundles (see figure 4.4). As mentioned this solution is not in line with the final vision where all static configuration of conditional compile type is removed.

### 3.4.3 Product inheritance

Products are defined by specifying a list of Bundles to include, but can also extend other products (see figure 3.19). This allows for abstraction and hierarchy in the configuration, making it possible to specify the differences between products rather than defining products from scratch. As shown in figure 3.20 a product can inherit a base product extending it’s set of Bundles. This means that a base product consists of the lowest common denominator of all the products connected to a platform cluster. To this base product, new functionality can be included by adding the desired Bundles.

### 3.4.4 XML syntax

The configuration file is designed in such a way that configuration concerning products and variants are moved into one file and configuration concerning modules to another one. In these two files XML syntax is used. The product file describes the products and in what way a product is inherited. All necessary Bundles are then pointed out by the product.
In the module file (that also uses XML-syntax), the Bundles are defined and legacy product configuration is placed. The rest of the configuration from the original configuration file is then lifted out to a legacy configuration file with the same structure as the original one. An overview of the proposed model can be seen in figure 3.21.

There is a great advantage in using XML syntax. Especially when used together with tools supporting auto complete which helps engineers to avoid making mistakes concerning naming standards and increases their overview of possible configuration choices. It also enables a more sensible error control of the syntax.

The XML-file is at compile time transformed into the old configuration interface format, allowing legacy tools to operate in the same way as today.

### 3.4.5 Hardware configuration

In the architects’ proposal the hardware configuration will eventually be realized as a dynamic module which the other modules at runtime can ask for information on the hardware properties of the device. For example, the camera module can at runtime ask the hardware module which camera chip that is available and choose the correct implementation.
Figure 3.21: Overview on architects new configuration structure.
Chapter 4

Improvement proposal

Bosch separates the implementation of a Software Product Line in the evolutionary and the revolutionary approach. As SEMC is a big company where the product line consists of millions of lines of code, a revolutionary approach is not likely to be implemented and the thesis will therefore focus on improvement proposals which can be implemented incrementally and evolutionary.

This chapter presents our proposal for a new configuration model. The proposal is a result of our studies in the evaluation of the current process. There is also an adaption of a recently developed configuration model proposed by software architects at SEMC 3.4, where we describe how this model can be improved using the concepts from our proposal.

4.1 Thesis workers proposal

There are two major issues with the current process discussed in the previous chapter. The first one concerns the lack of specifications. The second issue concerns the granularity of the configuration parameters, resulting in the configuration managers configuring low level features rather than products and developers configuring products rather than low level features. To cope with these problems we propose to introduce an abstraction layer in the configuration interface allowing a clear separation of product configuration and feature configuration. This will result in a clear and unambiguous responsibility distribution. We also propose that the Product Configuration Specification will be the one source for product differentiation specification, which will be used by all stakeholders in the development process, from product planners to developers and testers.

4.1.1 Abstraction level description

Requirements traceability

Today, it is hard to trace from variation points back to its requirements. A clear mapping does currently not exist and therefore a clear connection between the requirements and the variation points need to be introduced. We propose that the Configuration Packages discussed in section 3.2.4 should be used to associate the configuration parameters with the requirements. Because
the Configuration Packages will be used by the Product Planners to configure the products, it is natural that this structure should be used as a basis when creating the foundation of the configuration interface.

As mentioned before, there is a resemblance between the Configuration Packages and the Orthogonal Variability Model proposed by Pohl et al [PBvD05]. By associating the Configuration Packages with the configuration parameters this is further emphasized as the model is used throughout the development process with traceability links to both requirements and configuration parameters.

To make this division it is important that the Configuration Packages are defined with great precision and in an excogitative way. This must be done with a deep cooperation between developers and requirements engineers to fully capture all possible aspects of the variability, from requirements to implementation. We propose that this shall be done both when creating the Configuration Packages and when reviewing and refining them. When the need of new variability occurs during the implementation phase it is important that the new Configuration Packages are fully communicated back to the requirements engineers and documented in the appropriate way.

A full traceability between the Configuration Packages and the configuration interface will allow for an automatic generation of a product configuration directly from the Product Configuration Specification. This will save a lot of time spent on setting up new products and reduce the detective work associated with configuring them. Because the configuration is generated from the specification, a configuration audit of a product configuration will simply be an audit of the Product Configuration Specification. This activity is not even feasible with the current structure and complexity of the configuration interface. With this way of working setting up a new product, from the software point of view, will only be a matter of choosing the desired Configuration Packages (of which there are approximately 150) in the PCS. To achieve this kind of traceability we propose a naming standard which connects the configuration parameters to the Configuration Packages. This is discussed in section 4.1.1.

**Abstraction layer**

In figure 4.1 the current structure and the proposed structure are visualized. In the current structure the configuration file contains all detailed feature configuration on a very low level for every defined product. The configuration parameters are read in the software modules which are also specified in the configuration file. This file is edited by both CM and developers and because of its size and granularity it is vulnerable and subject to merge conflicts. In the proposed structure a new abstraction layer, \( CP\text{-Conf} \) (figure 4.1), is introduced between the product configuration interface and the software modules. The low level configuration is moved into this layer and a high level product configuration based on Configuration Packages are used on the product configuration level. This clearly separates the responsibilities between the developers and Product-CM. The developers naturally becomes responsible for the CP-Conf-layer and it’s associated modules. CM is only responsible for the high level product configuration.

To be able to introduce an abstraction layer, configuration parameters in the configuration file needs to be moved to separate files where parameters belonging
to a certain development team resides. The specification of selected modules needs to be in these files too, since it depends on the selected Configuration Packages. However, the definition of the module versions is a CM responsibility which is associated with the integration of delivered software modules. This therefore needs to be separated from the respective development teams configuration file. The model is visualized in figure 4.2.

This division removes a lot of false merge conflicts that is common today because the module versions are stored in the same file as the product configuration (section 3.2.9). The conflicts are false because the module versions in the configuration file delivered by the developers are not of interest to CM, since the correct module versions are determined from the module delivery system.

When this abstraction layer is introduced and the parameters are named according to the Configuration Packages, there is no need to change the existing naming of the variation points since they are moved out from the main configuration file.

Design of new configuration parameters

Today, the naming of the configuration parameters includes a feature description indicating what functionality the parameter affects. However, the features in the configuration parameters are not mapped to the requirements, and do not include a serial number connected to a requirement. This means that the only traceability that exists today needs human reasoning to connect the feature names from the two sources. Another disadvantage with the current name standard is that features are not the granularity for which the product planners configures the products.

We propose a new standard for configuration parameters where four types of parameters are available:

- The existing low level parameters which are used for product configuration today. To remove or change these parameters is an infeasible work.
The existing parameters which defines the hardware properties of the product (prefix CFG_HW). Today many of the parameters created are actually hardware dependent and could therefore be removed by using the hardware properties instead of creating new parameters. The syntax of the parameters should include the serial number from the hardware requirement specifying it’s value.

A new type of parameter for Configuration Dependencies. The name should include the dependency type (HW/FormFactor/Customer/Market). The syntax should be CD_<TYPE>_<NAME>.

A new type of parameter with a close connection to Configuration Packages by requiring the parameter name to include the Configuration Package serial number. The syntax should be CPM_<NBR>_<NAME>. This would allow for full traceability between variation points and requirements via Configuration Packages.

In an optimal solution the configuration interface for the products could be generated automatically from the requirements and Configuration Packages by the use of the above parameter types. This would remove much of the confusion regarding what a parameter affects and if it should be on or off for a particular product.

Unfortunately, there are some parameters that really are specific for each variant, that cannot be derived from hardware specifications and that are not part of a specific configuration package. An example is the acoustic settings which are fine tuned for each product. These should be considered as hardware configuration parameters and could maybe be associated with a requirement
stating that the product should be acoustically fine tuned. The actual acoustic information is kept in the file system rather than as static configuration.

For all special cases that appear, the following alternatives exists:

- The software that should vary is of a size that makes it meaningful to create a new Configuration Package. The package should be reviewed together with the requirements engineers and defined in the requirement management system.

- The software that should vary depends on high level hardware, form factor, customer or market. The Configuration Dependencies should be used. If new CDs needs to be defined, this should be done by the requirements engineers.

- The software is dependent on specific hardware. Hardware configuration parameters should be used to determine if software should be included or excluded. If such a hardware requirement is missing, it needs to be created either as a detailed hardware requirement or, if high level, as a Configuration Dependency (previous item).

- The software that should vary is small and none of the above techniques are appropriate. Variability using internal binding should be used.

Hardware configuration

Many of the configuration parameters associated with hardware are actually dependent on a few hardware configuration parameters. The values of these parameters can be determined by analyzing the higher level hardware parameters. An example is the configuration of the camera, which consists of about twenty parameters. These parameters are actually all dependent on the camera chip. In the same fashion as with the Configuration Packages, the high level camera chip parameter is left at the product configuration level, while its associated low level parameters are moved to the proposed abstraction layer and owned by the developers. This means that the developers does not need to consider which product that is to be created but only e.g. what chip that is used.

Documentation of configuration parameters

As mentioned earlier, the parameters are lacking proper documentation. There is a syntax for parameter comments in the modules, but it is not strict nor mandatory. By moving these low level parameters to the new abstraction layer the stakeholders are reduced to just the developers. Therefore the issue of lacking documentation only concerns them. To propose changes to the documentation of these parameters would probably not increase the quantity or quality of the comments. By introducing traceability between the product level configuration interface and the Configuration Packages, no further documentation is needed on the higher level. The name standard will be descriptive and in line with the Configuration Packages and will enable stakeholders to find more documentation in the requirement management system where the Configuration Packages are defined, described and associated with all requirements.
Responsibility of configuration

The low level configuration of Configuration Packages, which resides in the new abstraction layer, should be owned by the developers. Changes to the configuration of a Configuration Package are the developers responsibility.

The high level product configuration should be owned by the Product-CM. This does however not include the Product Configuration Specification, meaning that if the product configuration is automatically generated from PCS the responsibility of Product-CM is simply to control this generation and make minor changes to the result if needed.

The Product Configuration Specification should be owned by the Product Requirement Coordinator together with the Product Planners who uses the document to select the desired Configuration Packages for their products.

Today the Configuration Packages are owned by the requirements engineers. If the Configuration Packages are used as we propose the involvement from the developers has to be greatly increased when defining them.

Obsolete parameters

A great quantity of parameters in the configuration file are obsolete, meaning that they have the same value for all products in the product line, or that they have not been changed in a long time. Most of these parameters can not be changed without resulting in compile errors due to the evolution of the platform while the parameter has not been changed.

We suggest that the parameters that have not been changed over the last two platform clusters (approximately two years) should be moved to a file containing all obsolete parameters. This file should be locked for change. Parameters that do change but has the same value for all products should be moved to the development team’s private file and should probably not be part of any Configuration Package used therein.

4.1.2 Availability of product specification

Automatic configuration

All available Configuration Packages in the platform will be included in the Product Configuration Specification and a connection to the previously mentioned abstraction layer is made. This can be used to ease the automation of the configuration. If the high level configuration of products is based on the Configuration Packages and the name standard corresponds to the high level configuration parameters, the configuration file could be generated from the Configuration Packages defined in the requirements management system in an automated way.

If this is done, configuration of new products will be made a lot easier, which means that CM people can leave the basic configuration to the Product Planners and then only focus on the transformation from PCS to the actual configuration file. Of course, further investigations needs to be made to figure out how this tool will be implemented in practice.
Easy configuration interface

In the new process, the Product Planners are intended to do the basic configuration. Since their technical knowledge sometimes is limited, it is necessary to visualize the configuration in an easy way. Therefore, we suggest to introduce a user interface based on the Configuration Packages which should be possible to use without advanced technical knowledge. Here, benefits could be made by using the already specified dependencies between the Configuration Packages to control the possible choices of such an interface. Such dependencies could be visualized to the user to increase their understanding and awareness. Inspiration can be taken from e.g. web tools used in the car industry where more detailed features depends on more basic ones, like e.g. what car body, and what engine being chosen [Vol08]. In a similar way, Configuration Packages that are mutual exclusive can decide which further configuration choices that are available. Just like the prize tag can change when a car is configured, the defined license costs and memory footprint of a Configuration Package can be used to give the product planner information on the costs associated with the configured product.

There are possibilities for two different tools used to simplify and automate the configuration. The first tool could be an easy to use web application that lets the users create a proper configuration based on the Configuration Packages. The input to this tool will be the available CPs, their dependencies and the user’s input consisting of the CPs chosen for the new product. The output will be a Product Configuration Specification that is correct, complete and valid. The second tool will ultimately not need user interaction but would rather be a filter that transforms the PCS document into XML-based product configuration files.

In the case study from Bosch Gasoline Systems (section 2.3.3) the lack of tool support for product derivation was a major obstacle. Prototypes were developed, but they were not user friendly and required more effort to use than they could save. In the case study from Salion Inc (section 2.3.8) a commercial tool for product derivation was used successfully. However, this tool operates on the file level and is therefore not appropriate for the SEMC case.

Verifying configuration

If our proposal is implemented, a verification of the configuration against the requirements can basically be divided into two separate verifications. The first verification consists of verifying that the chosen Configuration Packages are the ones that are supposed to be in the product and that they have not been chosen in a way that contradicts the dependencies of exclusiveness of the CPs. The latter could be achieved by using these constraints in a tool when selecting CPs, as proposed in the previous section. The second part of the verification consist of verifying the contents of each CP against the requirements defined for them.

4.1.3 Managing configuration

As there are obsolete configuration parameters in the solution used today, there will in the future be obsolete Configuration Packages in the proposed solution. These CPs can be obsolete either by not being used in any product, and could
therefore be removed from the software product line, or by being used in every product, and should therefore not be variable but removed from the set of CPs and incorporated into the platform.

Ownership and deliveries of Configuration Package contents

Some of the Configuration Packages will have functionality that is shared between many development teams. At requirements level the requirements engineering teams have solved this by having one single owner for a Configuration Package. To ensure that these packages are properly configured and delivered, each Configuration Package should also have one single development team as owner.

Platform Packages

Since all requirements are associated with at least one Configuration Package, it is natural that all functionality in the software is also distributed into Configuration Packages. For functionality that should not be variable and rather always part of all products, this is misleading. The problem has been dealt with by having an attribute for the Configuration Packages specifying if it is a part of the platform or not, meaning that the Configuration Package is not possible to remove when configuring a product.

4.1.4 Process overview

An overview of the process with focus on product derivation and variability is visualized in figure 4.3. In the provide platform process, the requirement engineers define and manage the Configuration Packages with input from the developers, who also review the packages. This part of the process is already part of the improvements in the requirements engineering process. The developers use the Configuration Packages to bundle functionality, and performs the needed low level configuration inside the Configuration Package. The developers are responsible for this low level configuration and for the contents of the Configuration Package.

In the provide product process, product planners create the Product Configuration Specification based on the Configuration Packages defined in the provide platform process. This document is transformed into configuration syntax by Product-CM, resulting in a complete configuration of a product.

4.1.5 Implementation strategy (step by step)

To incorporate the proposal into the current configuration interface we propose an evolutionary approach where the implementation is done in small steps.

1. Create guidelines for all stakeholders (see section 4.3).

2. Move the configurations associated with each development team to it’s own file that is included in the main configuration file. The compiled result will still be the same, but merge conflicts and especially false merge conflicts (section 3.2.9) will disappear since each team has it’s own file and since module versions are kept in the initial file which is not merged but rather updated by Product-CM.
3. Include the new parameters for Configuration Packages, Configuration Dependencies and hardware specifications in a product specific configuration file, which is included into the main configuration file when that product is built. The compiled result is still the same.

4. Inform the development teams of the new way of configuring products. Give them the guidelines for product configuration and for management of CPs. Let them replace their configuration based on product names with configuration based on Configuration Packages, Configuration Dependencies and high level hardware specifications. Support them in this activity and set a date when all product dependent configuration should be replaced. Because both the old and the new way of configuring products can coexist in the same interface the transformation can be done iteratively with little impact on the development process.

5. Remove the product parameters so that they cannot be used in the development teams’ files anymore.

### 4.2 Adaption of the architects’ model

In this section we will describe how the software architects’ model (section 3.4) can be adapted using the principles described in the improvement proposal (section 4.1).

The architects’ idea of introducing Bundles containing collections of packages is closely related to the concept of Configuration Packages. This enables a good possibility to connect the Bundles to the requirements in a similar way as in the previous proposal. One of the greatest concerns will be to define a proper granularity of the Bundles.
4.2.1 Increase separation of product and feature configuration

The main objective of our proposal is to separate the product configuration and feature configuration in order to reduce complexity and clarify responsibility. The model proposed by the architects (figure 4.4) needs further separation between product and feature configuration to achieve this in a short-term perspective. This could be done by moving static product configuration from the product specific file into Bundle specific files, as shown in figure 4.5. In the beginning the Bundles would not be dynamical as in the final vision but would rather contain the Bundle specific configuration in a legacy configuration file. A major problem in the conversion to this structure will be that different Bundles are likely to share configuration between them. Because these dependencies cannot initially be solved by dynamic communication between modules, it is important that the inclusion of a Bundle also yields a configuration parameter associated with the Bundle, which can be used by other Bundles to determine if the Bundle is included. Since the XML-structure is compiled into the old configuration interface syntax, all parameters used at Bundle level can be seen as public.

The granularity of the Bundles must be carefully defined. If Bundles should be the only means for product differentiation in the future, the size of a Bundle needs to be big enough to be manageable for configuring products and at the same time small enough to support all differentiation that is necessary.

Since a company is required to pay for licensed applications and codecs included in the software, there is a need for excluding them in products where they are not used. This can be solved by turning each asset connected to a license cost into a Bundle. Though, it can be risky since decisions concerning which license Bundles to be included in a product can be too detailed for a Product Planner to perform. Especially if the amount of license assets reaches unmanageable proportions. In the definition of the Configuration Packages in the requirement management system there is support for tagging with both license costs and memory footprint which makes it possible to designate whether they are associated with additional costs or not.

4.2.2 Hardware configuration

It is not obvious if there should be e.g. one Bundle for the camera application or one Bundle for each available camera chip. From the modularization point of view, the goal should be that there is one Bundle containing the generic software of the camera application and one Bundle for each camera chip containing the specifics for the chip based on a common interface. The communication between these Bundles should be dynamical with loose dependencies. In the near future this solution is not likely and a transition solution based on static configuration is a less revolutionary approach.

In the short-term perspective the hardware configuration in the architect proposal is located in legacy product specific configuration files. Because these product files contains all hardware configuration for a specific product, it is subject to be changed by many development teams increasing the risk of conflicting changes. By dividing product and platform configuration into different files there is a risk of unintended overriding of configuration parameters. The
design does not separate the product and feature configuration and we therefore propose that the hardware configuration follows the same pattern as the other configuration (figure 4.5). By e.g. only specifying the hardware chip at the product level and then infer the more detailed hardware configuration at Bundle level, the separation of product and feature configuration is clear and the responsibility is unambiguous. This means that developers do not need to care for the product specific configuration files, but only for the detailed hardware configuration in the Bundle owned by the development team.

We propose that this level of static configuration is kept also in the long-term perspective. Since there are no hardware differences between different customers of the same product, the one-variant vision is still achievable with static hardware configuration. As mentioned in section 2.1.2, the complexity and cost of developing a system increases as the binding time is delayed [Sva03]. This motivates having the hardware variability kept as external variability which is bound at the pre-compile phase.

4.2.3 Clarify responsibility

In the architects proposal the responsibility division is a bit vague. Here responsibilities needs to be clarified to be able to make the working procedure more convenient. This can be accomplished by making a clear separation between configuration of products and of features, as mentioned earlier. Today, e.g. developers are involved in the configuration of products. Here, it is more desirable if the developers could put their focus on only configuring features and leave the product configuration to CM people.
To make sure that the process will continue to be stable, it is necessary that it is stated who to be responsible for defining, modifying and refining the Bundles. If this is not done things can easily degenerate, especially when the granularity of a Bundle is changed at a later stage in the developing process. Since defining the Bundles with high granularity is a basic condition to be able to configure new products in an efficient way, it is necessary for both requirements engineers and developers to define the Bundles together. This cooperation means that the importance of a clear responsibility division increases. However the final responsibility should not be shared, but rather be possessed by one of these roles. Our proposal is to make the requirements engineers responsible for the Bundles since this is according to the new requirements process that is on its way in. This in contrary to the architects proposal, where Bundles are created and owned by the developers.

As mentioned earlier, responsibilities for configuring new products are very unclear today. This also needs to be tied to a specific role, likely the Configuration Managers since they are the ones responsible for building the new products.
4.2.4 Variability linked to requirements

In the architect's proposal, issues concerning the connection back to requirements have not been taken into account in a satisfying way. The concept of Bundles and the concepts of Configuration Packages described earlier could be interlaced. We therefore urge a use of the already defined Configuration Packages as a basis when defining the Bundles. This is a good opportunity to establish and maintain traceability all the way back to the requirements in a similar way proposed by Pohl [PBvdL05]. We propose to use the name Configuration Package instead of Bundles since that will clarify the connection back to the requirements and since the term Bundle could be confused with terminology in Java.

To some extent, it can be necessary to redefine some of the existing Configuration Packages and even to create new ones to cover the proposed Bundles. When doing this it is also necessary to take into account that Configuration Packages are used side by side in the different clusters.

Our general opinion is that the possibilities to use this concept together with our proposal and the existing process are very good. Using this concept throughout the whole process in a consistent way will avoid duplicated work done by the different stakeholders. Therefore we suggest that this should be taken into account and that such a connection will be carefully elaborated. The Product Configuration Specification, based on Configuration Packages, can be used to generate the product configuration since it also contains hardware specific details and Configuration Dependencies.

4.2.5 Version control of Bundles

In the study of the current process it was found that configuration parameters are created for a variety of reasons (3.2.8). In order to support the usage of configuration parameters for version controlling and non simultaneous integration, the Bundles need to have version control capabilities. To support different configurations for the same code base, the version handling of Bundles needs to be kept separate from the usual version control facilities. This could be done by e.g. incorporating a Bundle version as an XML-attribute to the Bundle definition. This would allow for different versions of a Bundle to be used for different products in the same platform cluster. It would also make it possible for teams to deliver dependent changes non simultaneously and then upgrade the version number of the used Bundles when all software have been delivered. It should also be possible not to use version control of the Bundles. If no version is specified, the latest version will be used automatically. The main occasion for wanting to choose another version is during maintenance in a specific cluster. For these situations, it is important to be able to choose which version of a Bundle to be used.

4.2.6 Customer specifics

The architect's goal is to produce only one variant of each product. In the short-term perspective the customer-specific variants will still be produced as products that extends the generic product definition. This allows for separation of customer-specific into separate configuration files. These configurations then affects all of the customer's products. In a long-term perspective most of the
customer specifics should be moved from external binding (static configuration) to internal binding (dynamic configuration) to reduce the number of managed variants.

4.3 Create guidelines

To introduce the new way of working with variability and product configuration, information and guidelines needs to be produced for the different stakeholders and their respective activities. Where applicable, constraints could be introduced to avoid a wrong way of working.

When the Product Configuration Specification is used as a definition of the products actual configuration, the Product Planners and Product Requirement Coordinators will need guidelines for how this document should be updated and managed. If a tool is created for easier product configuration, guidelines for this tool is also needed. Because changes to this document will have great effects, the change process will need to be carefully defined.

When the Product-CMs are responsible for the actual product configuration file, guidelines are needed for how the PCS is transformed into a product configuration and what additional work is needed. The guidelines should also specify when the transformation should be done and how configuration changes that arise at CM level is propagated to the PCS.

The developers will create software and associate it with Configuration Packages. Guidelines are needed for how this should be done and how the detailed feature configuration should be achieved. When there is a need for increased variability or when the set of Configuration Packages does not match the software, the guidelines should specify how new Configuration Packages are created in a proper way.

The requirements engineers are the owners of the Configuration Packages and with the use of CPs throughout the development process the importance of a proper managing of these is greatly increased. The guidelines need to incorporate how the requirements engineers should communicate with the developers in the creation or change of CPs. When new CPs are created the developers also need a clear procedure for the technical setup in the actual configuration.

4.4 Managing Configuration Packages

As mentioned, a new process for handling requirements is already in place but not yet fully established within the company. In this process it is stated that the requirements engineers are responsible for creation, redefinition and removal of Configuration Packages. This process have defined procedures for doing this and this praxis should continue to be followed. According to the proposed changes the redefinitions of the CPs needs to involve concerned development teams as well. When functionality within a CP needs to be differentiable, either this have to be done by making it customizable or by breaking it out into a new CP. It is critical that CPs remains the lowest means for product differentiation in the static configuration. When lifting functionality out into a new CP it is also of high importance that dependencies to the origin package are defined and that affected development groups are counseled before the responsible team decides
to carry out the changes.

The functionality within a CP might also evolve over clusters. This means that a CP in a later cluster might differ from a previous one but still have the same name. This is important for e.g. Product Planners to be aware of when choosing CPs when new products are being set up.
5.1 Evaluation of improvement proposals

The evaluation of the improvement proposals was carried out as a desktop pilot, where the new structure was applied to the existing structure, followed by a presentation for the different stakeholders who then commented and filled out forms. The evaluation of the two proposals will be done as one evaluation since both of the proposals concerns the separation of the current configuration into Configuration Packages. The differences between our proposal and our proposed adaptations to the architects’ proposal are mainly of technical or structural nature.

The desktop pilot was run on a subset of the configuration belonging to two development teams. The Navigation team and the Messaging team were chosen for this.

Two developers from each team was chosen to participate in the redefinition part of the evaluation, but more developers were asked to answer the questionnaire (see section 5.1.3). The reason for choosing two developers for the evaluation activity was to make sure that a deeper knowledge would be available and that the discussion would be more manageable. The two developers was chosen based on their knowledge of the configuration parameters.

The questions in the evaluation form was based on the improvements and potential benefits discussed in our proposal. Before the evaluation, the form was tried out by personnel at the Product-CM department.

5.1.1 Problems and limitations in the evaluation

When performing this kind of evaluation it is difficult to cover all aspects. One should also be aware that this evaluation only take a few of the affected development teams into account, and therefore some important information and angles of approach might not be known. Furthermore, the amount of variation points that each team is responsible for, and to what extent these variation points are share with other groups varies. Therefore it is uncertain exactly to what extent each development team will be affected of the proposal in this thesis.

To perform an evaluation including all affected teams would be impossible to cover within the limits of this thesis work.
There has not been any evaluation for other stakeholders, like Product Planners and Requirement Engineers. Although they are not involved in the technical parts of the proposal, they are part of the process associated with the proposal and it is therefore a drawback not to have these stakeholders represented in the evaluation.

5.1.2 Redefinition of configuration

The Configuration Packages defined by the associated requirements engineers was used to group the existing low level configuration parameters, as described in the proposal. This was done with cooperation with the chosen developers mentioned in the previous section. When parameters could not be linked to an existing Configuration Package the developers had to consider defining new CPs, Configuration Dependencies or hardware requirements. During the activity the discussions was documented as on the interviews in the process evaluation (section 1.2.3). The following considerations came up:

- Packages needs to be complemented with an advanced version for greater differentiation possibilities.
- Some packages needs to define dependencies to other packages.
- The differences between some of the similar Configuration Packages needs to be described by the requirement engineers.
- One package may in the future need to be split into several packages, and one common package that does not offer any end-user benefits but that the other packages depends on.
- Problems may arise when new Configuration Packages needs to be created instantly. Communication with the requirement engineers will then be a bottleneck.
- There are packages which will not be possible to remove from the product due to strong dependencies. The product planners should not be allowed to deselect this package.

5.1.3 Evaluation form

After the redefinition of the configuration the developers was asked to fill out the evaluation form (section A.1) answering questions concerning the improvement proposal and it is possible benefits and drawbacks. The form was sent out to all seven people in one of the development groups and to 27 people in the other development group, covering half of the members. In the second group the selection was based on picking every other member from the team list sorted by names. 87\% have answered and the result is compiled in tables 5.1, 5.2, 5.3 and 5.4.
I have been involved in product configuration before. 7 8 1 6 6 2.9
I do not see any problems with the current way of configuring products. 8 7 10 2 1 2.3
Our team takes responsibility and ensures that our configuration is correct for all products. 0 6 7 12 3 3.4
The proposal is easy to understand. 0 3 5 14 6 3.8
The proposal is easy to implement. 0 3 14 9 2 3.4
The responsibilities in the proposal are clear. 0 0 11 12 5 3.8
The separation of product configuration and feature configuration is clear. 1 3 6 11 7 3.7
The proposal is likely to improve the quality of the products. 0 2 9 12 5 3.7
The proposal is likely to improve the efficiency of the product configuration activity. 0 1 7 15 5 3.9
The proposal does not affect my work negatively. 0 1 8 10 9 4.0
The proposal is overall a better solution than the current way of working. 0 0 5 17 6 4.0

Figure 5.1: Results from quantitative evaluation. The questions are answered by a 1-5 scale where 1="Do not agree" and 5="Fully agree".

<table>
<thead>
<tr>
<th>Question/Statement</th>
<th>Quant</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>What quality aspects are affected by the proposal?</strong></td>
<td></td>
</tr>
<tr>
<td>Configuration becomes more manageable.</td>
<td>3</td>
</tr>
<tr>
<td>Quality will be improved.</td>
<td>2</td>
</tr>
<tr>
<td>Less risk of misconfigured products.</td>
<td>2</td>
</tr>
<tr>
<td>Clearer responsibilities</td>
<td>2</td>
</tr>
<tr>
<td>No quality aspects affected</td>
<td>2</td>
</tr>
<tr>
<td>Many misunderstandings can be avoided</td>
<td>1</td>
</tr>
<tr>
<td>I don’t know.</td>
<td>5</td>
</tr>
<tr>
<td>&lt;blank&gt;</td>
<td>13</td>
</tr>
<tr>
<td><strong>How is your work affected from an efficiency point of view, in a short-term perspective?</strong></td>
<td></td>
</tr>
<tr>
<td>Not at all.</td>
<td>9</td>
</tr>
<tr>
<td>Slightly negative as some effort is needed for the change.</td>
<td>3</td>
</tr>
<tr>
<td>Efficiency will be affected negatively because maturity is always a concern in the beginning of a new process</td>
<td>3</td>
</tr>
<tr>
<td>Less work for developers.</td>
<td>2</td>
</tr>
<tr>
<td>Effort is needed for education</td>
<td>1</td>
</tr>
<tr>
<td>Efficiency will be improved</td>
<td>1</td>
</tr>
<tr>
<td>Very little, don’t spend much time on configuration today.</td>
<td>1</td>
</tr>
<tr>
<td>Displaying version tree of configuration would be faster.</td>
<td>1</td>
</tr>
<tr>
<td>&lt;blank&gt;</td>
<td>7</td>
</tr>
</tbody>
</table>

Figure 5.2: Results from qualitative evaluation
<table>
<thead>
<tr>
<th>Question/Statement</th>
<th>Quant</th>
</tr>
</thead>
<tbody>
<tr>
<td>How is your work affected from an efficiency point of view, in a long-term perspective?</td>
<td></td>
</tr>
<tr>
<td>Not at all.</td>
<td>8</td>
</tr>
<tr>
<td>Very little, don’t spend much time on configuration today.</td>
<td>4</td>
</tr>
<tr>
<td>Probably improve</td>
<td>4</td>
</tr>
<tr>
<td>Less work to maintain the configuration.</td>
<td>3</td>
</tr>
<tr>
<td>To some extent.</td>
<td>2</td>
</tr>
<tr>
<td>I don’t know</td>
<td>1</td>
</tr>
<tr>
<td>&lt;blank&gt;</td>
<td>10</td>
</tr>
<tr>
<td>What are the drawbacks of the proposal from your point of view?</td>
<td></td>
</tr>
<tr>
<td>I can’t see any</td>
<td>7</td>
</tr>
<tr>
<td>Important to consider handling of different languages and filesystems/customers.</td>
<td>2</td>
</tr>
<tr>
<td>Problems deciding what packages are appropriate.</td>
<td>2</td>
</tr>
<tr>
<td>Configuration Packages will be too large and can probably not offer the level of detailed configuration needed.</td>
<td>2</td>
</tr>
<tr>
<td>There will be initial confusion</td>
<td>2</td>
</tr>
<tr>
<td>None, but gain seems small.</td>
<td>1</td>
</tr>
<tr>
<td>If a quick fix is required and it is not possible without rearranging CPs, this could lead to a dirty fix which is hard to maintain.</td>
<td>1</td>
</tr>
<tr>
<td>Problems handling dependencies towards other groups.</td>
<td>1</td>
</tr>
<tr>
<td>Does not handle customer specifics.</td>
<td>1</td>
</tr>
<tr>
<td>The number of variation points will increase</td>
<td>1</td>
</tr>
<tr>
<td>I don’t know</td>
<td>1</td>
</tr>
<tr>
<td>&lt;blank&gt;</td>
<td>9</td>
</tr>
<tr>
<td>How does the separation of product and feature configuration affect your work?</td>
<td></td>
</tr>
<tr>
<td>No impact</td>
<td>9</td>
</tr>
<tr>
<td>Will be easier to change the configuration.</td>
<td>3</td>
</tr>
<tr>
<td>Very little</td>
<td>3</td>
</tr>
<tr>
<td>Better overview of the changes in the configuration.</td>
<td>2</td>
</tr>
<tr>
<td>Simpler for developers when not having to think about products.</td>
<td>2</td>
</tr>
<tr>
<td>Developers can focus on delivering a platform, not products.</td>
<td>2</td>
</tr>
<tr>
<td>Requires less involvement from Product CM</td>
<td>1</td>
</tr>
<tr>
<td>Harder to understand connection between feature and product</td>
<td>1</td>
</tr>
<tr>
<td>Little, since we already have the strategy of switching features and not products inside the modules.</td>
<td>1</td>
</tr>
<tr>
<td>&lt;blank&gt;</td>
<td>8</td>
</tr>
<tr>
<td>Do you see any problems in the dentition and refinement of the Configuration Packages together with the requirements engineers?</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>11</td>
</tr>
<tr>
<td>There will probably be a stabilization period until things are clear.</td>
<td>2</td>
</tr>
<tr>
<td>I don’t know</td>
<td>2</td>
</tr>
<tr>
<td>Yes, requirement engineers need to learn more about specific configuration packages.</td>
<td>1</td>
</tr>
<tr>
<td>There is a risk that the CPM-flags will just replace the current flags.</td>
<td>1</td>
</tr>
<tr>
<td>Must be done with care</td>
<td>1</td>
</tr>
<tr>
<td>The developers need more information about this concept</td>
<td>1</td>
</tr>
<tr>
<td>Possible problems when customer wants to change details in the configuration</td>
<td>1</td>
</tr>
<tr>
<td>&lt;blank&gt;</td>
<td>7</td>
</tr>
</tbody>
</table>

Figure 5.3: Results from qualitative evaluation, continued
How is your responsibility for product configuration (not feature configuration) affected by the proposal?

<table>
<thead>
<tr>
<th>Question/Statement</th>
<th>Quant</th>
</tr>
</thead>
<tbody>
<tr>
<td>My responsibilities will be smaller (in a good way)</td>
<td>4</td>
</tr>
<tr>
<td>Very little.</td>
<td>4</td>
</tr>
<tr>
<td>I don’t know</td>
<td>3</td>
</tr>
<tr>
<td>Not affected</td>
<td>3</td>
</tr>
<tr>
<td>Completely removed.</td>
<td>2</td>
</tr>
<tr>
<td>&lt;blank&gt;</td>
<td>11</td>
</tr>
</tbody>
</table>

Figure 5.4: Results from qualitative evaluation, continued

5.1.4 Evaluation with CM-strategists

To get a broader evaluation the evaluation form was adjusted (A.2) and sent out to two CM-strategists. When they had answered the form independently, a meeting was held where the answers were discussed. The result from the form and the meeting can be seen in tables 5.5, 5.6 and 5.7.

<table>
<thead>
<tr>
<th>Statement</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>I do not see any problems with the current way of configuring products.</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.0</td>
</tr>
<tr>
<td>The teams takes responsibility and ensures that their configuration is correct for all products.</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.0</td>
</tr>
<tr>
<td>The proposal is easy to understand.</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>4.0</td>
</tr>
<tr>
<td>The proposal is easy to implement.</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>3.0</td>
</tr>
<tr>
<td>The responsibilities in the proposal are clear.</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3.5</td>
</tr>
<tr>
<td>The separation of product configuration and feature configuration is clear.</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>4.0</td>
</tr>
<tr>
<td>The proposal is likely to improve the quality of the products.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>5.0</td>
</tr>
<tr>
<td>The proposal is likely to improve the efficiency of the product configuration activity.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>5.0</td>
</tr>
<tr>
<td>The proposal does not affect Product-CM’s work negatively.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>5.0</td>
</tr>
<tr>
<td>The proposal is overall a better solution than the current way of working.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>5.0</td>
</tr>
</tbody>
</table>

Figure 5.5: Evaluation with CM-Strategists, Quantitative. The questions are answered by a 1-5 scale where 1=”Do not agree” and 5=”Fully agree”.
<table>
<thead>
<tr>
<th><strong>Question/Statement</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>What quality aspects are affected by the proposal?</strong></td>
<td>There will be a process for product derivation, and someone will be responsible. Today there is no right or wrong in product configuration, since there is no defined process. Today there is probably much extra work because a good process and structure is missing. The connection to the requirements makes it possible to verify a configuration. Today we cannot assure any quality of the configuration. The configuration will be specified from the products, not the other way around. The quality of the requirements will be increased when they are used as basis for configuration. The requirements are unclear today. There should be a CCB for the requirements and the PCS.</td>
</tr>
<tr>
<td><strong>How is Product-CM’s work affected from an efficiency point of view, in a short-term perspective?</strong></td>
<td>Much work in the beginning, efficiency will decrease. Maybe the effort will not increase but will rather be shifted from merging and integration to work with the process and the actual configuration. Simpler integration. Fewer responsibility issues.</td>
</tr>
<tr>
<td><strong>How is Product-CM’s work affected from an efficiency point of view, in a long-term perspective?</strong></td>
<td>Early elimination of configuration errors should reduce the need for late configuration changes. We will have control over what we are doing, avoiding mistakes and wasted work. New variation points will not be created at product level if not planned. Late and unplanned changes to the configuration will be avoided. Product configuration will be less volatile and work will be easier to automate. It will be possible to start with a blank configuration rather than a copy, giving a clean configuration without obsolete settings. Less complicated merges, more manageable structure.</td>
</tr>
<tr>
<td><strong>What are the drawbacks of the proposal from your point of view?</strong></td>
<td>The configuration will be more stiff, flexibility is decreased. This may however not be a problem if the process for changing the configuration is clear and streamlined. If that is the case, changes will be less risky. The process must specify how late changes and exceptions should be handled, important that the process can always be followed.</td>
</tr>
<tr>
<td><strong>How does the separation of product and feature configuration affect your work?</strong></td>
<td>It makes it much clearer who is responsible for what.</td>
</tr>
<tr>
<td><strong>Do you see any problems in the definition and refinement of the Configuration Packages together with the requirements engineers?</strong></td>
<td>It is not clear that a Configuration Package can be mapped to one development group. Therefore the proposal does not necessarily remove manual intervention in handover from development group to Product-CM, and possible results in handover of feature configuration between development team. May be hard to get a streamlined process when many stakeholders are involved. When Configuration Packages are shared between groups, dependencies may be complex and integration work may be shifted to the development groups.</td>
</tr>
</tbody>
</table>

Figure 5.6: Results from qualitative evaluation with CM-strategists
| Question/Statement | | |
|--------------------|------------------|
| **How is Product-CM’s responsibility for product configuration (not feature configuration) affected by the proposal?** | It is now possible to take a responsibility for product configuration. |
| **What would the major wins be if the proposal is implemented?** | Less configuration errors and late changes. |
| | Possibility for easier and simpler integration. |
| | We know what we are doing and we know what is right and wrong. |
| | Possibility to verify configuration, today we only see that a test case failed. |
| **Is it possible to achieve new things/activities as a result of the proposal?** | Possible to verify configuration. |
| | Controlled changes. |
| | Possible to automate product derivation. |
| | Possibly auto-integration, but this is not addressed in the proposal. |
| **What would the costs be to introduce the proposal?** | In money, very little. |
| | Education is needed. |
| | Resources for process definition. |
| | The costs are not a concern in relation to the current situation. |
| **How large are the possibilities to automate parts of the configuration process in the future?** | Bigger, but it does not cover physical dependencies between development groups and Product-CM. |
| | See previous questions. |
| **To what extent do you believe that the stakeholders of the process will accept and support the proposal?** | Buy-in could be fairly good. And even better if it can be used to simplify deliveries from development teams. |
| | The developers would be positive if they do not have to care for product configuration. Demands from product projects can be forwarded to Product-CM and the developers will feel more confident. |
| | Very important to inform the developers of the new process. |
| | Product Planners could be slightly negative, because they will have less flexibility and cannot change their minds late in the process. |
| **How would you like Product-CM’s responsibility for product configuration to look like?** | Product CM should configure products, not modules. Product Configuration should be handled from a specification under change control and the SW configuration should be possible to test against the specification for correctness. |
| | CM could be responsible for the CCB of the Product Configuration Specification, but the requirement engineers, product requirement coordinators or product planners should be the owner of the documents. |
| | CM should be responsible for transforming the PCS to configuration syntax. |

Figure 5.7: Results from qualitative evaluation with CM-strategists, continued
5.2 Evaluation Analysis

From the evaluation it can be seen that the participants have been involved in configuration and that they see problems with how it is handled today. The proposal is fairly easy to understand, something that depends on how much information that was given to the participant. To get as many answers as possible the information was held short and concise.

On the question regarding ease of implementation, the result is slightly positive. Some mentioned in the qualitative part that customer specifics were not addressed enough. One participant also addressed the need for education in the short-term perspective.

Most of the participants thought that the responsibilities and the separation of product and feature configuration is clear. On the qualitative part this was confirmed by many statements saying that their responsibilities would be reduced.

On the question regarding product quality most answered 4, which could be considered as slightly better product quality. However, in the qualitative part it was mentioned that there is less risk of misconfiguring products. It is possible that the participants claim that the products have a high quality today and that this will therefore not be increased by the proposal. The quality of the products are though achieved by an inefficient and iterative procedure. The question addressing improvement in configuration efficiency scored higher, which indicate that this proposal would have a larger effect on efficiency in the way of working rather than end-product quality. From the qualitative part there was statements saying that the configuration would become more manageable and less time consuming, which emphasize this.

On the question regarding drawbacks there are concerns on the definition of CPs and their dependencies. It was mentioned that there will be a stabilization period until the CPs are clearly defined. They are expecting that quick fixes will be hard to handle using CPs, and that there will therefore be dirty solutions which are hard to maintain. There is a risk that the number of CPs could increase and that the same problems will arise again. Some are also worried about customer specific configuration, which the proposal does not specify in detail.

Most participants state that their work will not be affected negatively, and in the qualitative part they stated that there will be less work for the developers with the proposal. Their responsibilities will be smaller and some participants understood that their responsibility for product configuration will be completely removed. Overall they think that the proposal is a better solution than the current way of working.

In the evaluation with the CM strategists then responses were positive. Among the positive comments were the possibility to define a clear process with unambiguous responsibilities, possibilities to automate product derivation and verification and a more efficient product derivation process. The concerns regarded the need for a streamlined process for managing the Configuration Packages, including exception handling. Possible dependency problems when a Configuration Package spans many development teams were also discussed. Due to this it is possible that some of the integration problems that are currently at Product-CM are moved to the development teams owning the Configuration Package. The overall impression from the CM strategists were very positive.
Chapter 6

Future work and Conclusions

6.1 Future work at SEMC

The thesis suggests how the variability can be managed and connected to the requirements, but it does not fully specify a process for product derivation since this would be of less general interest. Therefore SEMC need to define a process for managing Configuration Packages at the configuration level and deriving products from the specifications available.

When the changes suggested in this thesis are further established and a process is followed consistently, tools can be inferred to ease the management of the process. As we see things, these possibilities concerns tools used to generate configuration directly from the requirements management system and an interface connected to the requirements engineering system used to ease the choice of Configuration Packages. It is of high importance that such an interface is designed in a clear and user friendly way since it will be used by people with less technical knowledge than engineers, e.g. Product Planners. We suggest something similar to a web-interface, where Configuration Packages easily can be chosen with regard to dependencies and mutual exclusions.

Other improvement areas of the overall configuration is to go from static configuration to more dynamic configuration (3.2.7), as far as possible. This transition is actually one of the intentions from the software architects at SEMC at the moment. The goal is to only use static configuration (conditional compile) when creating products, and to use dynamic configuration when creating variants. Configuration of interface availability is planned to be achieved by the use of mechanisms like IDL and CHAPI who allows components to communicate in a language natural way.

6.2 Future research

There are quite much literature on the subject of Software Product Lines. Since the companies that use a product line approach are relatively large, more detailed research may not be of general interest. Case studies like this one can
however help companies to identify problems similar to their own and get inspiration from the solutions presented. It is therefore important that the case studies are detailed enough to cover all problems that might be of interest to other companies.

The lack of tool support for product derivation was mentioned in some case studies, but again it is hard to use off-the-shelf products for this kind of activity in a large organization with an existing product line that is not adapted to the structure of the tool. For organizations about to transition into a product line approach, the commercial tools may be of interest but from what we have seen there is not much competition on that market. For that purpose further research in this area, possibly based on case studies like this, may be of general interest.

6.3 Conclusions

This thesis have been investigating product derivation from a Software Product Line at Sony Ericsson Mobile Communications. This have been done in the form of a case study where interviews and document studies have been used to collect data according to stakeholders perspectives and to gather knowledge about how product derivation and configuration is actually carried out today.

In this case study it have been found that new products are derived by copying the configuration of the most similar previous product and by iteratively configuring the product in a cycle between developers, CM and test (see figure 3.18). The variability is not clearly specified and documented and the responsibilities are unclear. There is no connection between the requirements and the configuration possibilities in the product line. These aspects affects the possibilities to verify the configuration and the time spent on product configuration.

To be able to cope with these issues, improvements have been proposed consisting of an abstraction layer in the configuration interface. This abstraction separates low level feature configuration from high level product configuration and establishes a traceability from requirements to configuration. This is done by using existing Configuration Packages from the requirements engineering process. To clarify the product configuration and ensure that everyone is working consistently, we propose that a product specification based on these Configuration Packages is used throughout the company.

These are the identified problems and the corresponding solutions:

- **Large number of variation points with an unmanageable granularity.**
  Variation points are encapsulated into Configuration Packages, separating the high level configuration from the low level configuration and resolving the granularity issues.

- **Unclear responsibilities and undefined process for the product configuration activities.**
  By dividing the configuration into different layers and proposing a new way of working, responsibilities are clarified.

- **No clear traceability between configuration parameters and initial requirements.**
By introducing an abstraction layer based on Configuration Packages, the configuration will be clearly and directly connected to the initial requirements.

- **No complete product specification available.**
  A new and managed product specification based on Configuration Packages should be spread throughout the organization and used by all stakeholders.

- **Products are configured in an inefficient and iterative process without using the initial requirements.**
  By the use of a complete product specification and a configuration interface based on the same Configuration Packages, the configuration can be done at an early stage, maybe even automatically.

The evaluation shows that the developers are coherently positive to this proposal. To ensure that no major problems would appear, the changes were simulated together with two development teams. This showed no major obstacles, but emphasized the importance of cooperation between the requirement engineers and the developers in the definition of the Configuration Packages. The most important unsolved issue concerns customer specific configuration. This have not been the focus of this work and therefore concerns the developers.

The adaption of the architects’ model using our principles shows that the proposal is on a level which is of a general interest and can be applied to different contexts. The adaption has already been incorporated into the next cluster project at SEMC, which we also see as a confirmation of the relevance and maturity of the proposal.

The expectations with this proposal is to reduce effort and time spent on iterative configuration, to ensure a higher product quality by the ease of verification, to state a more clear responsibility among stakeholders and to make the information concerning variability within the company more accessible.

It can be stated that practicing software product lines improves the quality of the products and reduces the time spent. However, managing a product line and it’s variation points efficiently requires a consistent way of working and clear responsibilities.

There are some big challenges concerning the ability to maintain the new way of working. It is important that the Configuration Packages only reflects the current needs for variability and that Configuration Packages are not created proactively in the same manner as variation points are created today. It is also important to educate people in order to convince them of the gains achieved following the new praxis consistently and to make them understand the new structure and be aware of the responsibilities connected to the different stakeholders.

As the research questions stated (section 1.2.1), the contributions of this thesis includes a detailed investigation on product derivation from a large software product line. The problem situation has been described from different stakeholders perspectives and improvements to the current way of working has been presented and evaluated.
Bibliography


Appendix A

Evaluation forms
A.1 Evaluation form - Improvement proposal

We are performing an investigation of the current way of configuring products as a thesis work. The work includes an improvement proposal which we would like you to answer some questions about, as an evaluation of the proposal.

Today products are configured using a large configuration file containing the configurations of all products. This have become unmanageable due to the granularity and number of configuration parameters.

Our proposal involves a separation of the configuration into two layers, where one layer is more abstract and contains product configuration while the other is more detailed and contains feature configuration. At the same time we propose that this abstraction is aligned with recent improvements in the requirements engineering process, where detailed requirements are bundled into more general Configuration Packages. These Configuration Packages are defined by the TWG together with FG and are meant to be used by the Product Planners to configure new products. We propose that these Configuration Packages will also be used in the configuration as visualized in figure A.1. In the same manner, detailed HW-configuration can be derived from e.g. camera chip which is defined at product level. This separates the responsibility of the configuration so that developers do not have to care for product configuration but rather configuration within the Configuration Packages.

![Diagram](image_url)

Figure A.1: Example of new structure based on Configuration Packages.

For your team the following Configuration Packages are defined so far:

- <list of Configuration Packages>
The evaluation form consist of a quantitative and a qualitative part. The questions in the quantitative part are answered by a 1-5 scale where 1=“Do not agree” and 5=“Fully agree”:

<table>
<thead>
<tr>
<th>Statement</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>I have been involved in product configuration before.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I do not see any problems with the current way of configuring products.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Our team takes responsibility and ensures that our configuration is correct for all products.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The proposal is easy to understand.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The proposal is easy to implement.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The responsibilities in the proposal are clear.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The separation of product configuration and feature configuration is clear.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The proposal is likely to improve the quality of the products.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The proposal is likely to improve the efficiency of the product configuration activity.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The proposal does not affect my work negatively.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The proposal is overall a better solution than the current way of working.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The qualitative part of the form consists of the following questions:

- What quality aspects are affected by the proposal?
- How is your work affected from an efficiency point of view, in a short-term perspective?
- How is your work affected from an efficiency point of view, in a long-term perspective?
- What are the drawbacks of the proposal from your point of view?
- How does the separation of product and feature configuration affect your work?
- Do you see any problems in the definition and refinement of the Configuration Packages together with the requirements engineers?
- How is your responsibility for product configuration (not feature configuration) affected by the proposal?
### A.2 Evaluation form - CM-Strategists

The evaluation form consist of a quantitative and a qualitative part. The questions in the quantitative part are answered by a 1-5 scale where 1=”Do not agree” and 5=”Fully agree”:

<table>
<thead>
<tr>
<th>Statement</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>I do not see any problems with the current way of configuring products.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The teams takes responsibility and ensures that their configuration is correct for all products.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The proposal is easy to understand.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The proposal is easy to implement.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The responsibilities in the proposal are clear.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The separation of product configuration and feature configuration is clear.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The proposal is likely to improve the quality of the products.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The proposal is likely to improve the efficiency of the product configuration activity.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The proposal does not affect Product-CM’s work negatively.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The proposal is overall a better solution than the current way of working.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The qualitative part of the form consists of the following questions:

- What quality aspects are affected by the proposal?
- How is Product-CM’s work affected from an efficiency point of view, in a short-term perspective?
- How is Product-CM’s work affected from an efficiency point of view, in a long-term perspective?
- What are the drawbacks of the proposal from your point of view?
- How does the separation of product and feature configuration affect Product-CM’s work?
- Do you see any problems in the definition and refinement of the Configuration Packages together with the requirements engineers?
- How is Product-CM’s responsibility for product and feature configuration affected by the proposal?
- What would the major wins be if the proposal is implemented?
- Is it possible to achieve new things/activities as a result of the proposal?
- What would the costs be to introduce the proposal?
- How large are the possibilities to automate parts of the configuration process in the future?
- To what extent do you believe that the stakeholders of the process will accept and support the proposal?
- How would you like Product-CM’s responsibility for product configuration to look like?