Master's Thesis

Variability Management with a Feature Perspective

David Karlsson, D00

Department of Computer Science
Lund Institute of Technology
Lund University, 2004

ISSN 1650-2884
LU-CS-EX: 2004-04

Variability Management with a Feature Perspective

(External version)

by

David Karlsson

Master thesis
Computer Science and Engineering

Department of Computer Science
Lund Institute of Technology
Lund, Sweden

February 2004

Table of contents

1 Introduction

Ericsson Mobile Platforms AB is a company that develops platforms for mobile communication. The customers of these platforms are mobile phone manufacturers, which use them to create salable consumer products.

To be able to compete on today's market for mobile platforms it is essential to be able to create a large variety of products. This has lead to the adoption of the *software product-line* approach since it enables developers to maximize reuse by creating (or instantiating) products from a common code base by means of configuration.

The configuration process is of great significance since it is one of the last steps before a product is delivered and it influences many areas of the development process such as build time and software testing.

Creating a multitude of products from a common code base creates difficulties of its own. There is no standardized way of modeling that captures the software variabilities and commonalities in a clear, structured fashion. Furthermore, creating a multitude of products from the same assets usually has effects on the architecture of the system, which has to be recognized and handled in a satisfactory manner.

The mobile platform customers have very high demands on functionality and configurability. It is challenging to foresee future customer requirements and how they may affect the software architecture, by introducing new variabilities. This makes it a necessity to fully understand and to have a complete process for managing variabilities.

This thesis addresses both the design and implementation aspects of software variabilities in the context of the configuration process.

The product-line approach and variability is briefly described in chapter 2.

Feature modeling is introduced in chapter 3, as an approach for handling software variabilities in the design. It is a modeling approach where a customer perspective on variability is used and it enables the developers to get and overview of the variabilities of a software system.

Chapter 4 describes a solution for reducing unnecessary dependencies related to configuration. A new mechanism for configuration is also discussed.

In chapter 5, tool-support for tracing dependencies is discussed. Areas for further investigation are also discussed in the conclusions.

1.1 Problem description

The problems, which this thesis addresses, are described here together with necessary clarifications.

1.1.1 Background

The EMP platform is a source system (product-line) from where a number of configured platforms can be derived. The source platform is a scaleable and configurable product. Configurability is very important for the platforms customers because they need flexibility to create the products they desire.

An important part of the development of the source platform is to identify, analyze and describe the commonalities and variability among product-line instances. The variability is explicitly modeled.

A *variation point* is a concrete point where variants of an entity can be inserted. A variation point delays design decisions. There are dependencies between variation points. A more detailed description of this is given in chapter 2.

There are different mechanisms that can be used in order to implement variability at different binding times (source time, compile time, link time, install time, …, run time). There is a dependency between variation points and architecture components (modules). The implementation of variation points can also increase the dependencies between modules.

Pre-processor directives are one mechanism that is used in the EMP platform in order to include or exclude functionality.

1.1.2 The problem - Configuration dependencies

When creating a customer specific platform, the software is configured to include or exclude functionality based on what the customer has purchased.
In many cases it is difficult for the programmer to track the implications of different configurations. There is a need to model different types of dependencies:

- Dependencies between variation points
- Dependencies between variation points and modules
- Dependencies between modules due to the implementation of variation points

Ways of reducing these types of dependencies should be studied. Tracing and documenting these dependencies is also important. The difficulty lies in the often-conflicting goals of minimizing memory use and constantly increasing the functionality.

1.1.3 The tasks

Through discussions with parts of the Software Architecture group, the scope of this thesis was defined by four different tasks.

- Study configuration dependencies in EMP mobile platforms.
- Perform a trade-off analysis of alternative solutions for minimizing these configuration dependencies.
- Give a proposal on how the different types of dependencies can be represented during the modeling.
- Develop a tool to visualize configuration dependencies in the software platform.

1.2 Remarks

The scope (given above) of this thesis was too large for one person, which was evident fairly quickly. The main reason for that was the proposed tool construction, which is very complicated, and time consuming even if one just constructs a prototype.

Important as tool support might be, the other three assigned tasks had to take precedence since they were meant to increase the understanding and reduce the difficulties with configuration dependencies. This had to be done before considering tool support.

1.3 Approach

A major part of the work needed for writing this thesis consisted of analyzing theories about product-lines and variability. This was partly because of the theoretical nature of the thesis and partly of the fact that feature modeling and variability management is a relatively new field in computer science. It was therefore important to have a fundament of knowledge in this field to be able to conduct analyses and propose changes and improvements.

There are many different models and theories about how to handle variabilities and features but since they are relatively new it was important to have solid information about their use in real development projects. And that proved to be quite difficult since companies are not inclined to publicize detailed papers publicly. But to be able to provide at least some information about the state of things in other companies regarding variability management was a major issue.

Some minor program construction was also done for parsing and analyzing the platform code base.

In the following chapter product-lines will be described and discussed.

2 Software Variability

This chapter is focused on concepts that are important for describing and handling software variabilities.

2.1 Product-lines

A software product-line is a set of software–intense systems sharing a common, managed set of features that satisfy the specific needs of particular market segment or mission and that are developed from a common set of core assets in a prescribed way[5].

EMP develops software platforms, which are used by companies for creating salable mobile phones and other wireless devices. Developing platforms for mobile systems is very costly and it is unrealistic for most manufacturers to actually create their own platform from scratch.

The market for mobile phones is very diversified, there are a number of different customer groups that demand different mobile phones with different functionality and this is reflected in the software development process of telecommunication companies.

The usual approach to software development is to develop one product at a time but that is impossible due to market demands. It is essential to be able to provide customers with platforms for the entire spectrum of mobile equipment, from the cheapest low-end to the feature packed high-end products. And that is impossible to do if all the different products are developed in completely independent projects because that would create immensely expensive products.

This creates the need for a product-line. Instead of developing the different products independently one exploits the commonalities in the products, different as they may be, they are all used for mobile communication and there are great similarities between them.

A code-base or a source platform is developed and maintained. The platforms are created or instantiated from this code-base through configuration using various mechanisms.

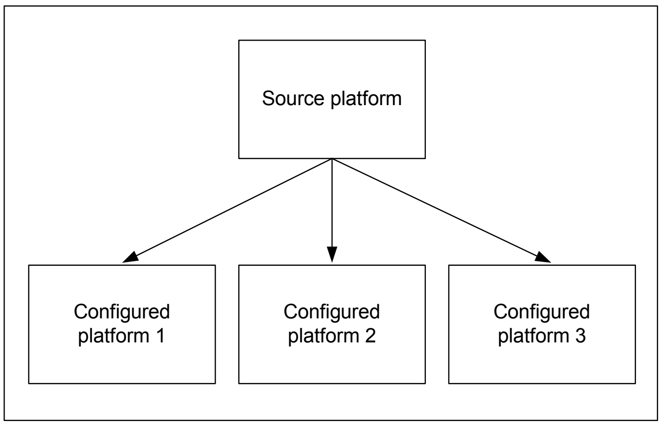

Figure 1.

The platform is then customized for the specific needs of the customer. The saleable platform is created by configuration and customization, but the customer also configures the platform to create a saleable product, e.g. a mobile phone.

The product-line approach promises to enable an organization to create a large variety of similar products, through a planned form of reuse, but it demands a lot from the development process.

There is a price to pay for the increased throughout put of different products, that the product-line approach allows. And this has to be recognized and handled to maximize the benefits.

2.1.1 Product-line problems

The two main problems with product-lines are modeling and managing variability. It may seem simple to develop a source platform and then configuring it to create the different platforms. However, the idea with product-lines is to rely heavily on reuse of software components. But in many cases it is not enough just to pick and choose components to instantiate a product. The different components will also need to be configured. This means that the internal structure of the software components will be complex.

The goal is to make it possible for product builders to focus on specific platforms and reuse the common architecture and building blocks that make up the source platform.

However, the complexity of the code-base makes it difficult, in many cases, to actually understand the implications of different configurations. This complicates testing.

If the implications of a configuration are difficult to understand, it will be hard to know what needs to be tested, when instantiating a new platform. And testing all possible configurations is not always a feasible solution.

Maintenance is also affected by these complexities. By maintaining the code-base you are maintaining all the products that are created from it. Changes (e.g. bug fixes) to the code base are done only once, instead of once for every product, which is a big advantage. But different customers want different things from different platforms. A change in a platform, if not customer specific, should be integrated into the code-base, which may affect many other platforms. The initial cost for making a change is less but understanding its effects might be challenging. The evolution of software components further complicates this since different versions will have to be maintained concurrently.

Handling and understanding the complex relations that are a result of the variability in a highly configurable platform is essential for product-lines. This demands a structured and well-developed configuration process, which describes the configurations, the variation points and the constraints and dependencies between them. Feature modeling (described in Chapter 3) is one approach for managing variability in the modeling domain, and it creates a view of variation points and their dependencies.

2.2 Variabilities explained

A variation point is a point in the design domain where a decision is delayed. Delaying design decision creates a more generic product since each choice taken limits the set of products that can be created.

There is an important distinction between the variation point and its implementation. A variation point is a point in the design, where a decision is delayed (like the use-case below). The implementation of that variation point is in the code. And the implementation is not a point; it does not need to be localized in one place. There could be a number of statements like the one below, in different files, but it would still be the implementation of one variation point if the flags MP3 and WMA are responsible for the decision on what variant to choose.

E.g. Use-case: “the user should be able to play audio in some format”

Example of an **implementation** of a variation point at run-time:

```
if(MP3)
  Variant 1 code
else if(WMA)
  Variant 2 code
end
```

In this case there is a choice between running either the code from variant 1 or the code from variant 2, depending on what flags are set or not.

When the decision is made concerning which variant to use, it is said that the variation point is bound. Variation points can be implemented (and bound) at different binding times, e.g. pre compile-, compile-, link-, install- and run-time (like above).

3 Feature modeling

“A feature model covers the commonalities and variabilities of software family members, as well as the dependencies between the variable features”[2].

The main idea with the product-lines is to use core assets to create a variety of different products. This created a need for ways of modeling the variabilities and commonalities this approach creates. The most influential approach for doing this is feature modeling, which was introduced in the Feature Oriented Domain Analysis (FODA) feasibility study [2]. A recent extension to this model is the Feature-Oriented Reuse Method (FORM) model [3].

In the following sections feature modeling and its usage will be described. Feature modeling is interesting because it gives developers a new tool for describing the variabilities and commonalities in a software system. Use-cases are commonly used for this but it is in many cases difficult to describe software systems clearly using just this approach. Especially in feature intense systems, like telecommunications software, a feature oriented view has proved to be valuable. Examples of telecommunications companies, that are using concepts influenced by FODA, are Nokia[4], Telecom Italia[6] and Nortel[12] and Bell Northern Research[13].

3.1 Modeling

This section defines features and motivates the usage of a feature model.

3.1.1 The value of feature modeling

"The feature model serves as a communication medium between users and developers"[2].

One of the ideas with the feature model was to capture the variabilities and requirements that the customer recognizes and to be able to represent them in a way the customers can understand. It could therefore be used in marketing as a way of describing and presenting the system's configurability.

However, the greatest advantage is for the developers. Feature modeling will give developers a clear overview of the features in the system and how they depend on each other. Variation points and constraints between features are explicitly modeled which promotes a more structured handling of variability and configuration.

Feature modeling is a part of many (application) domain-modeling approaches. Domain modeling is complete approach for identifying and describing the variabilities and commonalities within a family of applications. If feature modeling is going to be used an evaluation of these approaches should be done since feature modeling is more useful in conjunction with a domain modeling approach. Evaluating these approaches is outside of the scope of this thesis but a good start would be to read about the FODA inspired model FeatureRSEB[13] which is a simpler and more understandable model than FODA itself.

3.1.2 Defining a feature

A feature is a characteristic of a system that is relevant to a stakeholder. In other words, something that someone e.g. a customer recognizes and values. This is one of the more commonly used definitions; the vagueness of this definition is both its strength and weakness.

When discussing features in general terms this definition serves its purpose. But, when designing something specific it would be valuable to further specify what the word feature means, and define different classes of features. A feature is not just a chunk of functionality.

In the FODA model a feature is either

- Mandatory
- Optional
- Alternative

Optional means that there is a choice whether the feature is bound or not, as opposed to mandatory. Alternative means that only one feature can be chosen from a set of features.

There can also exist constraints between features:

- Requires
- Mutual exclusion

A feature can require the existence of a set of features. The existence of a feature may also be mutually exclusive to the existence of another feature.

3.2 Feature diagrams

An important part of feature modeling is the feature diagram. Two examples of feature diagrams are given below, with different graphical notations to show the different constructs present in the feature model. The value of the different notations is also discussed.

3.2.1 FODA notation

A feature can either be optional, mandatory or alternative. An optional feature is described with a small circle and an alternative is represented with an arc, all other features are mandatory (none in Figure 2). Mutual exclusion and requirements are not modeled graphically in the FODA notation.

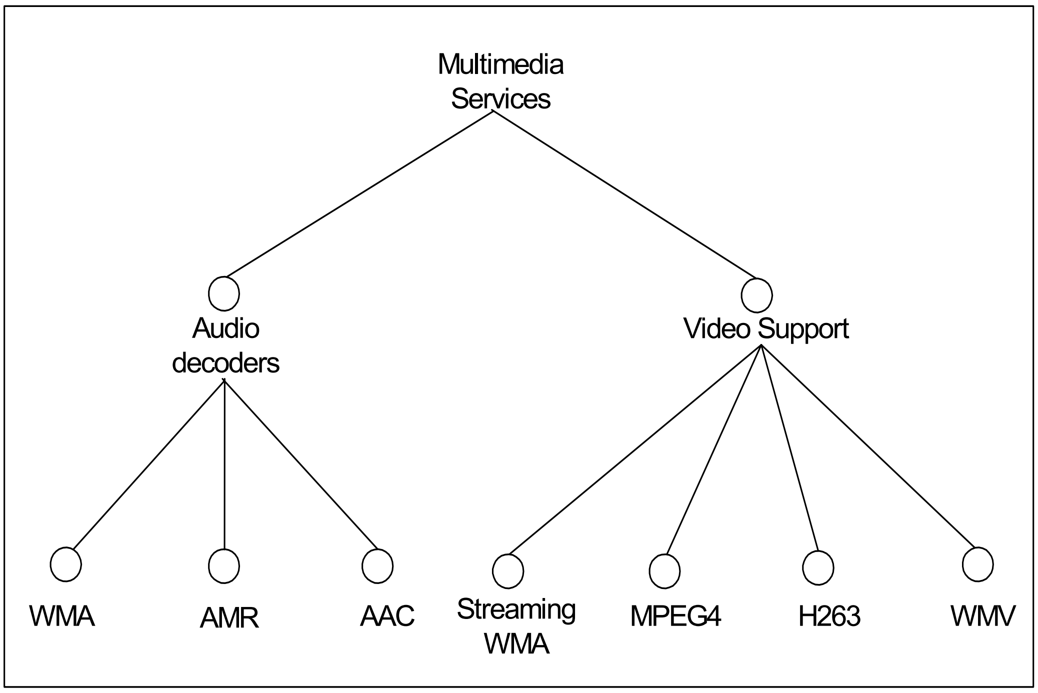

Figure 2. An example of configurable multimedia services

The interpretation of this diagram is very straightforward: all features are optional, which is true, but it is an oversimplification because requirements are not modeled graphically in FODA.

This example only uses features controlled by compiler flags. The features could be further broken down to show internal configuration. However, just documenting the usage of a specific mechanism is not feature modeling. Variability is most easily captured when pre-processor flags are used but that does not mean that one should only model variability implemented with that mechanism.

3.2.2 UML notation

Extensions of UML[7] for modeling features has also been proposed, which appeals to developers that are more familiar with UML notation. They have also given the notation more expressiveness, compared to the FODA notation. The same example is given using UML notation in Figure 3 below. The extension in expressiveness consists of modeling requirements graphically. The feature IPv4 was included because streaming WMT requires that it exists.

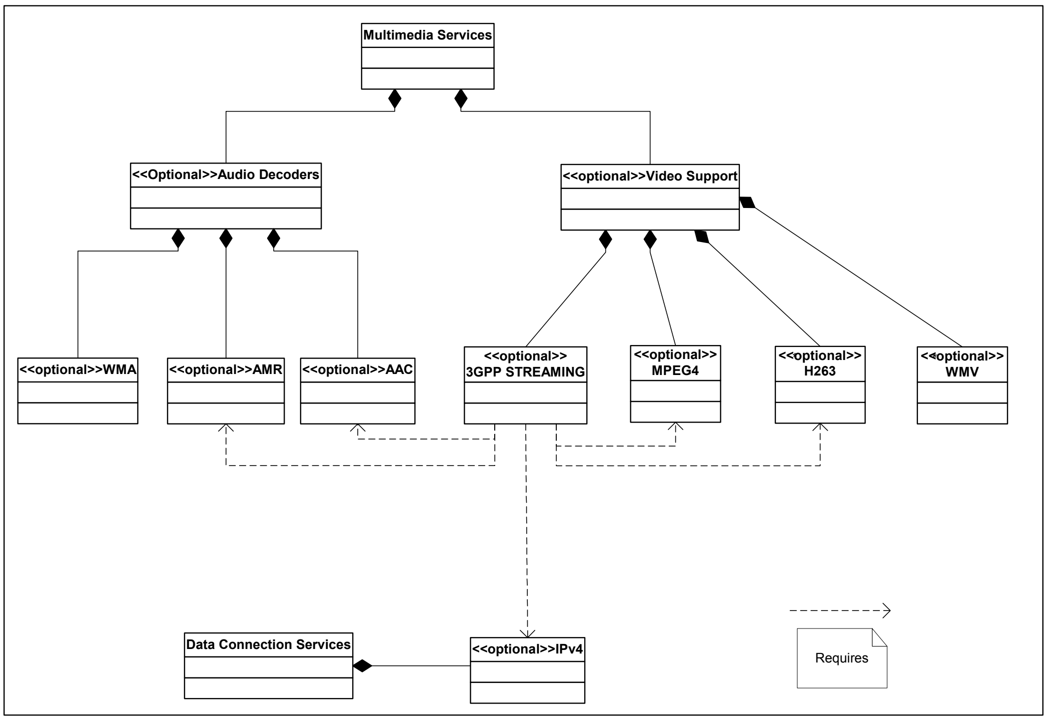

Figure 3.

This gets complicated very fast, even thou only requirements from "3GPP STREAMING" are modeled in this example. The complete picture is much more complex. Dependencies like these are modeled in the feature model but a decision was made (in the FODA model) not to include them in the graphical notation because they make the diagrams too complicated to read.

So creating a new graphical representation with more constructs is not something that necessarily is more useful. This does not mean that these requirements should not be captured, they should, but they might not need to be modeled graphically, at all times. Textual descriptions of the features should in all cases accompany the graphical model.

3.3 Feature selection

Feature modeling is usually done after domain modeling but describing the complete process is outside of the scope of this thesis. Nonetheless, creating a first feature model of a system should be possible to do for domain experts, without a formal domain analysis. Hence, even though FODA is not incorporated in the development process, feature modeling could still provide an interesting and helpful view in the design and development process.

To create a feature model the different features in the system have to be identified, and how they depend on each other. To ease the task of extracting features FODA specifies four different categories of features.

- Capabilities, what an application can do, from the users perspective
- Operating environments, in what environment the application is used, different hardware, operating system etc.
- Application domain technology, e.g. domain specific methods
- Implementation techniques, e.g. mechanisms, protocols, etc.

This should ease the process of finding features and will also give the feature diagrams a layered structure with Capabilities being the highest layer and Implementation the lowest.

A good starting point for finding features and building the feature model is the use-cases, since that is where most variation points exist. An approach similar to the one given in FODAcom[6] (and extension to FODA) is to create the feature diagram by creating a root node of the use-case. The feature tree is then created by putting the root node in a composed-of relationship with the variation-points and then put the variants under their corresponding variation points.

An example where this approach is applied on a use-case:
(ME: mobile equipment)

Play Audio

The ME user listens to audio typically stored on a memory card or in internal flash.

The ME supports internal loudspeaker and accessories for both mono audio as well as high quality audio (stereo) via a handheld audio accessory.

The ME supports MIDI, MP3 and AAC in high quality. It is also possible to play mono AMR clips. If a stereo headset is used, the sound is played in both speakers.

Using this use-case and the approach described above will give a diagram similar to the one in Figure 4.

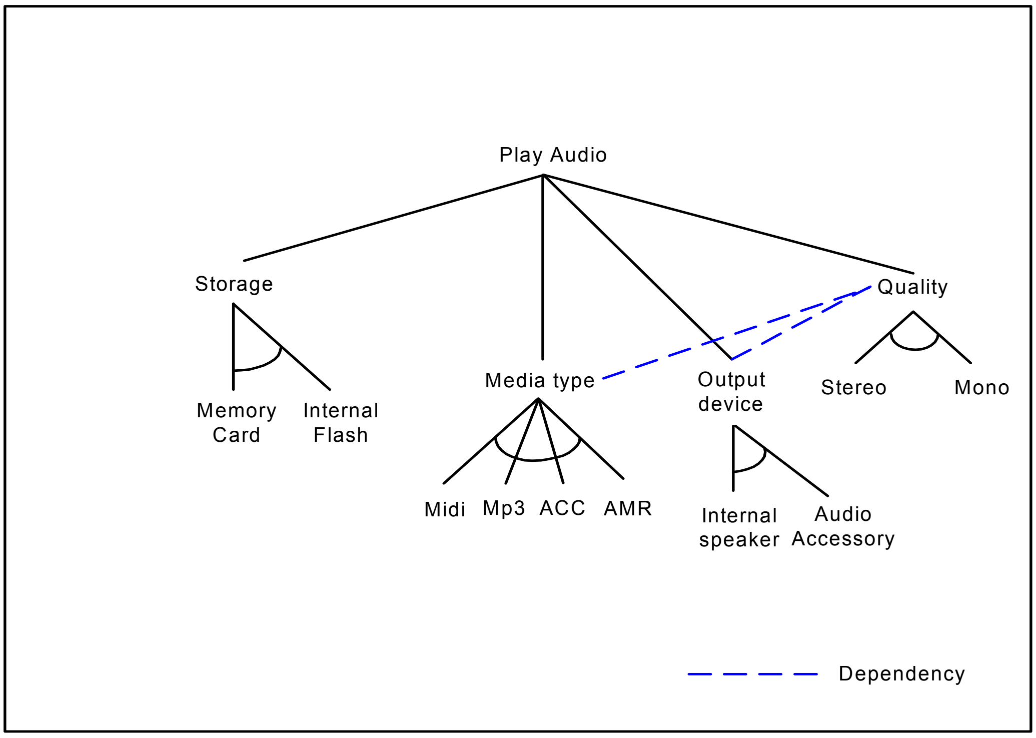

Figure 4. The diagram describes the variability seen from an end-user perspective when all features have been included. End-user means run-time, so media type, storage and playback device is a run time decision.

The interpretation of this diagram is when the user plays audio it is played exclusively on the internal speaker or an audio accessory. The audio is stored exclusively on a card or flash and the format of the audio is one of the supported media types.

How the quality attributes are incorporated in the model is not trivial because there are dependencies between the media type, the output quality and the output device. And it is not entirely clear which dependencies are present just from this use-case, e.g. is MP3 playable through the internal speaker? Yes of course, but that can not be deduced from the given use-case alone.

To be able to describe the constraints imposed on features it may be necessary to look beyond a specific use-case. But that is only natural because the feature model is not supposed to describe the use-cases but instead the features and their constraints and dependencies. Dependencies may very well exist between features in different use-cases. The use-cases can be a starting point for building a feature model, but it should be further developed by incorporating features extracted from other use-cases and requirements. And that is the advantage, since it gives you an overview of the features, which use-cases do not permit.

An issue concerning feature modeling is that there is a need to describe the variability in different steps. When the platform is instantiated the customer has a choice about which features to include or exclude. But the end user will also have a choice concerning which feature to use at runtime. Since the customer and end-user choice may be the same (but different at binding times) it will prove to be cumbersome to represent it in the same diagram.

E.g. media support for MP3 or AAC is bound at pre-compile time. If we have decided to move some of the configuration to runtime, and not to allow the end-user play MP3 (but still include support for it), that would mean that there is no choice concerning what audio codec to use, for the end-user. But there is still a choice whether to enable support for it in the platform.

Just because there is a choice concerning a set of variants that will be supported there is not necessarily an end-user choice regarding which one that should be used. This was not a realistic example, but scenarios like this could potentially create problems if different binding times are mixed in the same diagram.

3.3.1 Generating Feature diagrams

Relying on that developers will actually draw these diagrams and maintain them might be dangerous because they are not a necessity but a complement to use-case modeling. Motivating developers to maintain it will be difficult. Furthermore, there is probably not much room for adding additional costs for documentation. It is therefore important to have an intelligent approach for handling the feature model.

If structured textual feature descriptions are incorporated in the configuration process (which is described in Section 3.4) they will describe the system variability in a structured fashion and be used for generating feature diagrams. Feature descriptions are versatile and creating feature diagrams is just one application, which would be a motivator for maintaining them; hence the generated feature diagrams are more likely to be up to date.

One advantage with the FODA feature diagram is that they are easy to generate from textual descriptions, because of the simple tree structure. In the UML case you will need more complicated layout algorithms to handle the more complex structure, that e.g. requirements creates.

An example of using XML for generating UML diagrams is given in [9], where the problem of layout is addressed in a way which is satisfactory for most situations but there will definitely be cases where the diagrams must be modified to make them readable. This has to be taken in account, if a diagram generator is to be constructed then a diagram editor should accompany it.

You might of course decide to use the UML notation without using the requirement construct but then the only real difference will be a bigger and less manageable diagram.

There are cases where either solution is appropriate but what is of importance is not how features are represented but instead that they are formalized. Structured textual descriptions of the features, which describe the features themselves and the constraints, are necessarily.

3.4 Domain specific languages for feature modeling

One of the more interesting ways of handling feature modeling is to do so using a domain specific language[1] to specify and describe the features of a software system. The idea is to create a structured description for each feature in which you describe the feature itself and constraints and dependencies that affect the feature. In the table below an example of a possible feature description is given.

Key	Description
Feature description	A textual description, understandable to the user or customer and/or a link to documentation in e.g. HTML
Parent feature	The feature that this feature is a sub feature of
Feature Type	Optional, mandatory or alternative
Constraints and dependencies	All types of constraints on this feature, e.g. which other features are required to exist (or not)
Modules affected	Which modules are affected by this feature in the implementation
Binding Mechanism	The type of mechanism used to bind the feature e.g. Compiler flag, and which specific binding mechanism used
…	…

The "Modules affected" tupple would be the first step on an attempt to create a mapping from the feature domain to the implementation. This is a challenging task but it would create means to understand how modular dependencies are affected by variability, which would be interesting from a developer perspective.

Tool support for the creation and usage of this is a must, the textual descriptions should therefore be in XML because of the vast array of existing tools for parsing XML.

```
<FEATURE>
 <FEATURE_NAME>Feature name</FEATURE_NAME>
 <DESCRIPTION>
 …
 </DESCRIPTION>
 <PARENT>Parent name</PARENT>
 <FEATURE_TYPE>Optional</FEATURE_TYPE>
 <CONSTRAINTS>
  <REQUIRES>…</REQUIRES>
  <REQUIRES>…</REQUIRES>
   </CONSTRAINTS>
< MODULES_AFFECTED>Module id< /MODULES_AFFECTED>
</FEATURE>
```

If these structured descriptions are created there are at least three interesting uses, which are connected with each other.

- Using the descriptions to generate feature diagrams or other types of documentation, that can be used by developers and end-users.
- Creating configurations.
- Validating configurations.

3.4.1 Documentation and diagram generation

The first point is self-explanatory and easy with the descriptions in place, being so easy to generate there is no reason for drawing them manually. Especially in a large system it is unrealistic to think that someone will do so. Furthermore, feature diagrams are not always be the best form of representing the features; just a table with the feature descriptions might be suitable in some cases, which is just as easily generated.

3.4.2 Creating configurations

The whole idea with describing the features is to capture all the possible choices that can be made when a user is configuring (binding features). If it is feasible to capture all the possible choices then that can be described in the feature domain. If there is a way of transforming the configuration from the feature domain to the implementation, an actual configuration could be generated. By choosing a set of features, a set of e.g. compiler flags could be generated, that will give that specific configuration.

3.4.3 Validating configurations

The idea with validating configurations is to use the feature constraints to validate the configurations with a mathematical algorithm, in the feature domain. It would be helpful to do so if configurations are to be done from the feature domain. This way making sure that the configurations are valid before using them. An implementation of this is described in [1]. There are some improvements and additions that have to be made before it can be used in practice but they are addressed in[1].

3.4.4 An implementation

A prerequisite for implementing this is that it is possible to model most features in practice. It is relatively easy to do with features that are bound at pre-compile and compile time because of the mechanisms used; meta language and pre-processor flags, which usually results in distinctly defined features.

However, at later binding times parameterization is used, which creates less distinctly defined features. But, if capturing the features is too difficult, one could settle with showing that certain features are configurable and with what mechanisms, instead further decomposing them into smaller parts.

Given a code base that is already in place someone will have to describe all the features with feature descriptions. Tool support is needed in order to ease that process, and the textual descriptions will have to be maintained and synchronized with the use-cases and requirements. Using more or less abstract features could moderate the cost for this.

In any case, if structured textual feature descriptions are to be used it is important to take full advantage of them. Integrating them with a CM tool is a natural idea for minimizing the synchronization problems between the descriptions of the variability and the implementation of it. One way of doing this would be to add feature information into the description files and handle variability in such a way that the feature descriptions could be generated from the implementation, but this is something that needs further investigation.

The main interest in this chapter was how a feature model could be maintained. Creation of configurations from the feature domain and integration with CM tools is interesting, but something for the future.

3.5 Feature crosscutting

Product-line architectures have many advantages but its goal to maximize reuse also creates problems. The problem arise from the fact that features in many cases are not mapped to single components, but instead spread over a set of components which is commonly referred to as *feature cross-cutting*.

This may be a result of poor decomposition of the features into components, which could be a result of the fact that some features are not removable when initially introduced, but customer requirements make it necessary to sometimes exclude them. This makes it important to have clear configuration requirements, to avoid quick fixes as much as possible.

Feature crosscutting degrades the systems architecture but it seems to be unavoidable in complex systems because it allows developers to introduce new features with relatively small overall costs.

The cost for introducing a feature which crosscuts may be very low if it exploits commonalities with other features. In other words if it is reusing existing components or parts of components. Furthermore, if a feature has large commonalities with existing features a complete separation will mean that essentially the same thing will need to be maintained twice.

Nonetheless, feature crosscutting has to be treated with much care. There must be large amounts of commonalities between features to make it beneficial for them to crosscut. Even though it seems to be ineffective to maintain separate components which exhibit large commonalities, merging them might drastically increase the maintenance cost because of the complexity of the resulting component. Testing, readability and traceability will suffer to some extent in most cases, even if there were great commonalities between crosscutting features.

The difficulties lies in understanding when to decompose crosscutting features into separate components and when not to. It is difficult to quantify the different measures that determine if feature decomposition is beneficial but two fundamental properties of a feature is the amount of commonality with other features and localization; how widely the feature code spread in source files, over source files and over modules. Localization is important since it determines how easily it is to understand the impacts of a feature on a software system, which is a major difficulty for crosscutting features. And this could affect the configuration process, since it may be difficult to fully understand the consequences of a configuration.

There are no simple solutions but when using compiler flags localizing the usage at all levels is essential, especially inside single files. Using multiple flags in multiple places in C-files makes them very difficult to understand. Each compiler flag should therefore be made to mean as much as possible, not being used as much as possible. If a feature is to be removable it should, if possible, be done with one reference to one flag for each file where the feature is defined.

Feature crosscutting is acceptable if it means that distinctly defined code localized within files and components is spread over a set of components, since that will make the development and maintenance cost low enough to justify the structural problems it introduces. Furthermore, complex logical pre-processor statements should preferably be avoided since it decreases traceability and readability.

4 A Trade-off-analysis – Different forms of configuration

In this chapter, alternatives for reducing the number of unnecessary complete rebuilds and increasing the traceability of configuration are evaluated. An alternative configuration mechanism for SDE is also discussed in Section 4.2.

SDE is a software development environment, which handles the building process, and interfaces with configuration management tools.

4.1 Configuration alternatives

This chapter describes different approaches for replacing global configuration files.

4.1.1 Alternative A

If global configuration files are to be removed the information they contain has to reach the modules in a different way, and since global inclusion is removed the location of modules with interest of specific flags has to be explicitly defined. Further more, it must still be possible to do the configuration in one location.

For this to be possible it has to be specified who has interest in what flags to avoid creating unnecessary dependencies. This could be done in a special file (configuration dependency file), which would contain the flag names and the locations of internal module configuration files, which reference specific flags.

A generator could then generate internal configuration files for the modules, thereby giving them the feature knowledge that they need, according to the configuration dependency file.

Templates of the internal configuration files have to be created, which would mean creating files with tags where the defines-statements are to be inserted. Templates are necessary since it enables developers to insert static pre-processor directives in the configuration files.

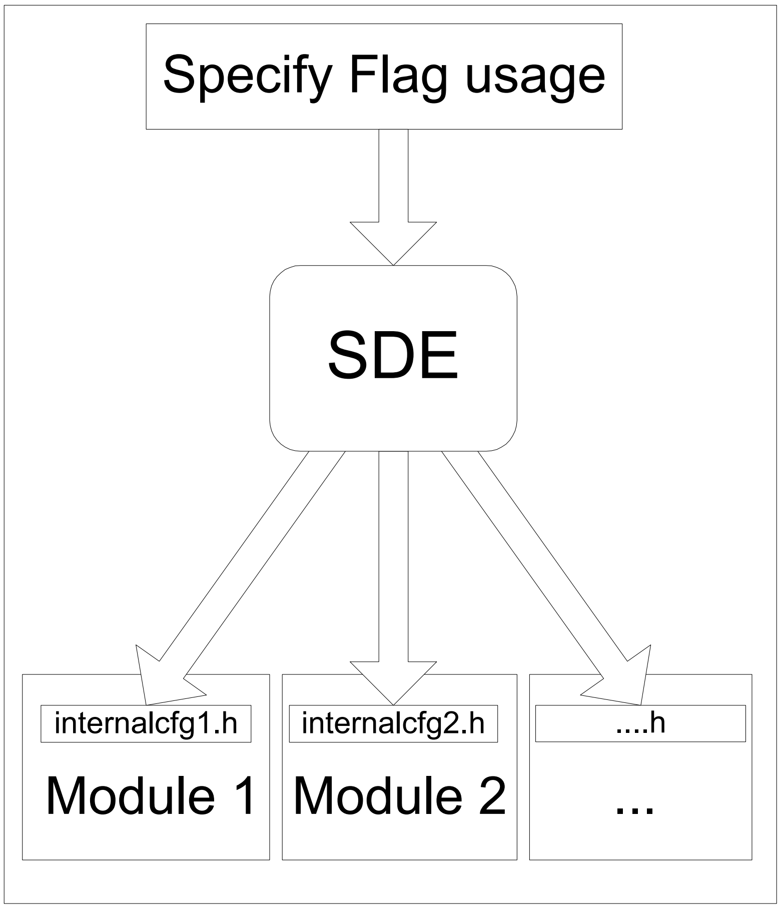

Figure 6. Conceptual outline for generation of module configuration

After the configuration files are generated they will have to be made available to the modules. Storing them locally and making the modules include them can do this. Another approach would be to generate a separate file, only containing the generated flags, making each internal configuration file include its specific flags. This would remove the need for tags in the files, but on the other hand, it would create an additional header-file for each module.

In any case, what is generated has to be compared with what was previously generated because it cannot just replace the set of generated files with a new set of generated files, even if they are identical. Otherwise this may lead to a complete rebuild if the build process is uses dates to recognize changes.

4.1.2 Advantages

- A flag change will only lead to recompilation of affected modules.
- Transparent to most end-user.
- Does not create additional header files (if an internal configuration file exists)
- Module configuration completely contained in the module.

4.1.3 Disadvantages

- The configuration dependency file, which describes what modules need what flags, has to be defined and maintained.
- Modules that are using the flags must have an internal configuration file.
- A tool for configuration is necessary. The dependency file gives you a centralized view of the configuration but changes must be done at module level.

4.1.4 Alternative B

A seemingly simple way of replacing global configuration files is to rely entirely on compiler switches. This can be done by creating module specific variables (in the meta language) that contain all the flags a specific module is allowed to reference. These flags can then be transformed into compiler switches, which are applied to the modules internal configuration file.

The compiler switches can then be used to control which internal flags that will be defined in each module. Hence, the result will be just the same as using global configuration files.

4.1.5 Advantages

- A flag change will only lead to recompilation of affected modules.
- Existing mechanisms are sufficient, modifications of CM-tools are in most cases not necessary.
- Does not create additional header-files (if an internal configuration file exists)

4.1.6 Disadvantages

- Adding a lot of complexity to the products description files.
- Modules that are using the flags must have an internal configuration file.
- Configuration is not contained in the module, but instead in the make file.
- Tool support will be necessary.
- The description of what modules need what flags has to be defined and maintained.

4.2 Discarded alternatives

In this section alternative solutions are presented, together with arguments why they were discarded.

One solution would be to create a single header-file for each flag and let the modules, which are interested in a set of flags, include the corresponding header-files. However, adding a large number of header-files is not a very beneficial solution since it will increase build time. It would also create the need for a tool support to give the users a centralized view of the configuration.

Another solution would be to create a header-file containing all the flags that are referenced by each functional stack since in most cases features do not crosscut over different stacks, thereby limiting their scope. Or putting them in groups, which are only allowed to reference certain modules. But there are no guaranties that the flags will not crosscut different stacks or groups in the future.

Maintaining this will be complicated since there are ideal groupings of the flags, in the sense that unnecessary dependencies are minimal. Adding or removing flags might make it necessary to change the grouping, to keep the unnecessary dependencies at a minimum. This will involve adding/removing files, include statements and pre-processor statements.

Hence, keeping the system buildable will inevitably be more costly than before, and tool support for configuration will also be needed.

4.3 Evaluation of configuration alternatives

Alternative B adds a lot of complexity to the platforms description files and maintaining them will require tool support. Furthermore, storing the configuration as compiler switches in the make file makes it difficult to understand what happens when the configuration is made, as opposed to in alternative A where the configuration is generated and stored in the modules internal configuration files.

Alternative A might demand that the CM tools are modified but it does not only reduce the number of rebuilds but also creates a clearer connection between the configuration and the implementation. This is valuable for the understanding of configurations and their implications, and would give more control over the usage of pre-processor flags.

The downside is that it will demand more work to keep the system buildable. The configuration dependency file must be maintained so that modules that require knowledge of certain flags get access to them. This should preferably be done by one organizational entity as opposed to by all module developers, to make sure that there are no unused flags in the platform, thereby preventing unnecessary complete rebuilds.

Whether Alternative A is a solution of practical use depends on a number of things. The value of the increased traceability and (the possible) reduction of the number of complete rebuilds should be compared to the implementation- and maintenance-costs.

4.4 Architectural changes

The discussion in the previous sections was focused on the replacement of global configuration files and even though it could be beneficial, it is a solution with little concern of the bigger picture. In this section a broader perspective is taken on how modules are configured in CM tools. However, this chapter is mainly concerned with ideas and not solutions, since further investigation is needed.

It is common that modules have knowledge of the implemented features since it enables them to determine how they are to be configured. An advantage with this is that each component can make sure that it is consistent with the rest of the system. Still, making software components very much dependant on their environment creates a number of problems. It is difficult to understand the dependencies between components, the system becomes less scalable and building separate parts of the system is complex.

4.4.1 Source tree composition

An approach that could have positive effects on architecture and issues mentioned in previous chapters (such as build time and traceability) is *source tree composition* [10]. The idea is to build a source tree of the system where the building blocks are configurable components. The components are configurable through interfaces which describe what the component provides, and also specify what it requires, thereby promoting a weak coupling between the components.

Features, which is a natural choice of top nodes in this source tree, points out and configures the components (or sub features) implementing it, and the components do likewise with their sub components.

The big difference between this approach and more commonly used approaches is that the requirements are directed solely downwards. A module does not require that a feature is active, but instead the feature requires a module to be configured in a certain way.

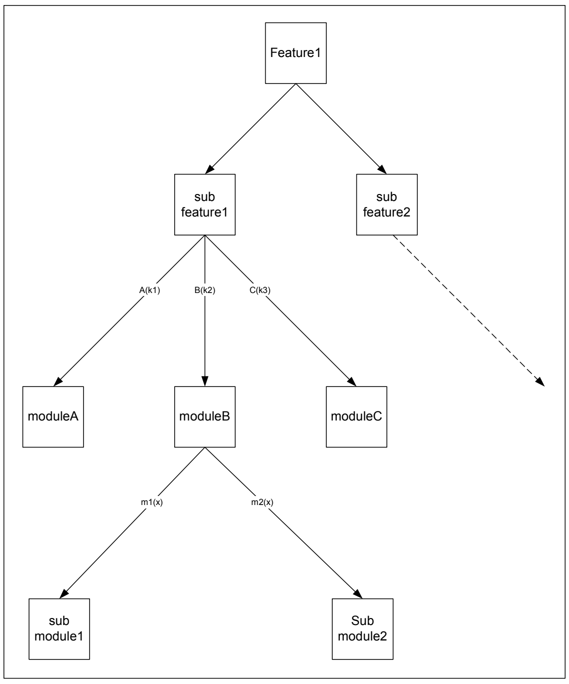

Figure 7.

To be able to build this source tree there must be a definition of each component containing its configuration interface, requirements and identity. In [10] this is called a source code package. With this in place, the idea is to pick the features and then let a tool collect the packages that are necessary for creating a consistent system with the correct build order.

When software components are configured by higher-level components partial configuration is necessary, because features sometimes crosscut over the same components. Otherwise that may create conflicting configurations, which has to be discovered and handled by the build process.

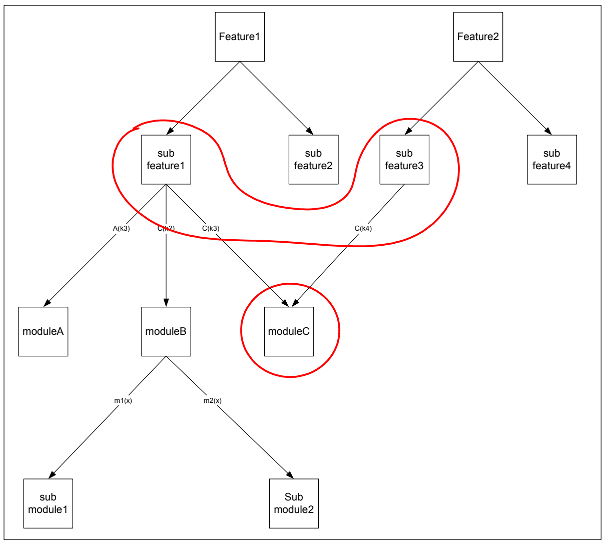

Figure 8.

In Figure 8, two sub-features require the same module and there is no reason to assume that they require the same configuration of module C.

A conflict occurs if two configurations are applied to the same component if, and only if, common parameters are bound to different values.

If two configurations are conflicting the build will be inconsistent and this has to be discovered to abort the build.

There might of course also exist dependencies between the input-parameters (which should be avoided) that can cause conflicts, but that has to be handled internally by the component.

When the packages are collected the source trees of the individual packages can be collected to create the complete source tree. And since the packages are collected according to the specification of requirements in the source packages, the correct build order will be implicitly given. The source tree will be built level-by-level down-to-top, since a component should only be built after all the components it requires are built.

4.5 A new mechanism for configuration of modules

Tools that support global variables as the single mean for configuration of modules are not ideal in terms of supporting traceability. A more restrictive configuration mechanism would also simplify the usage of configuration interfaces.

When a module is included, the name and the version are specified in the description language. It would be natural if the configuration of the module, in terms of a set of variables, were specified here too. The module would have to specify a configuration interface, e.g. what types of variables it accepts, where the type could be determined by the names of the variables since different data types are not supported.

It is very important that configuration descriptions of the modules are easily accessible for the users to make the configuration straightforward. The configuration description should at least contain a short textual description of the module, possible input values and the explicit requirements that the module imposes on the system.

4.5.1 Bundles of modules

Just changing the configuration mechanism will not remove the modules dependency on the features. The logic that is concerned with features has to be moved out of the module.

One could place it in the product description file but the risk is that it will make the product description file unmanageable, and not just difficult to read. It will also be much harder to distribute the responsibility of keeping the system consistent. In [10] it is suggested to create a module for each feature, which will contain the configuration logic and point out the modules that make up that feature. But it is probably sufficient to place them in a single description file, thereby making it possible to get an overview more easily and to avoid creating a large set of new files, that would have to be versioned and managed.

4.5.2 Breaking down features into functionality

To get a clear separation between the features and the implementation, the modules should not be allowed to reference feature flags (giving them feature knowledge). The feature flags must be broken down to implementation flags (one-to-one or one-to-many). This has to be done outside the module, either in the product or the feature layer. However, it is not always possible to do a meaningful functionality breakdown.

E.g. a pre-processor flag controls whether a feature is included or not. The feature is crosscutting over a set of modules, which means that the pre-processor flag is referenced in a set of modules. If the modules should not have feature knowledge the feature flags will have to be replaced with module specific flags.

If possible, the feature flag should be replaced with one module specific flag for each module. But it is not certain that the feature flag excludes/includes distinct functionality completely contained in single modules, because of crosscutting. This means that it will be difficult replace the feature flag without disclosing the feature name. The meaning of the code exclusions in a single module may not make much sense if it is not clear what feature it is a part of, which forces us to give the module feature knowledge.

One could also try to break down a feature flag into a number of flags (for each module) so that each flag would describe what it is actually used for. Even if the usage of a feature flag does not make sense in the context of that specific module it could be possible to decompose it to parts, which could be given meaningful names themselves.

However, this should be done with care since the result might be source files where a number of flags with different names are used, but the only combination will be either all are defined or all undefined, since they all replace a single feature flag.

In some cases the best solution will be to create module specific flags, which will reveal the feature name. This will give them feature knowledge, but it will still be an improvement because the connection between features and modules will be clearer.
Nonetheless, dividing features and implementation as much as possible would make the process of product instantiation more structured and therefore more easily understood.

4.5.3 The build process

Adapting to source tree composition is a major task but one could take a first step in that direction by decoupling the modules from the system by changing the way they are configured. That would have positive effects by itself; a more layered structure, and open the door for source tree composition without making it a necessity to actually do it.

This form of configuration would also make it much easier to create tools for tracing the configuration dependencies. Impact analysis on different configurations could thereafter be performed more easily.

The advantages are that there is a clearer separation of software components; requirements are modeled explicitly and requirements are always (or mostly) directed downwards.

5 Tool support

In this Chapter tool support for tracing configuration dependencies is discussed. A modification of SDE is suggested to increase the readability of description files. A simple tool which was used while writing this thesis, for simplifying flag searches is also described.

5.1 Tracing configuration dependencies

In the previous chapters much has been said about the lacking visibility of the implications of configurations. A solution to this could be to create a tool for increasing traceability.

The main problem is that most of the configuration is done by meta languages; the SDE description language, and C-pre-processor statements, which are alike in the sense that they have basically the same logical constructs.

The ideal would be if it was possible to create a map of the usage of these constructs in description files and the source files, but this is extremely difficult. This is mainly because the complex logical statements that these meta languages allow creates very complex structures. Considering the number of flags, files and references to these flags, the set of possible paths of execution is huge, and describing this clearly and informative is very difficult.

There are few commercial tools that handle this in a satisfactory manner and the research in this field is limited. In [15] an approach for visualizing the usage of pre-processor statements is described, which probably is the best attempt I have seen. However, this approach is only applicable for single files. Which is by no means sufficient.

But [15] gives at least the reader an understanding of how difficult it is to capture these dependencies clearly. Constructing a usable tool for doing this in an industrial project will require much more research.

5.1.1 Increasing the understanding of configurations in description files

Even though it is difficult to clearly describe the implications of a configuration in all stages, improvements can be made which would drastically increase the readability of the description files.

The description files are used to generate the make files. When the make files are generated SDE first pre-processes these files and exclude statements, depending on the configuration.

It would be very instructive to show which statements are excluded, thereby showing the implications of a certain configuration. The user would be able to see which files and modules that are actually included and what variables that are set, in a build.

5.1.2 Implementation aspects

To create this view it is necessary to pre-process single files and compare them with the originals, thereby deducing which lines are excluded.

A mechanism for pre-processing single files exists, and is used in a plug-in to Visual Studio. When this function is used on a description file the given file is pre-processed and returned with the extension _preproc.

The output format of the SDE pre-processor will have to be slightly changed.

- All variable assignments (in all description files) are evaluated, removed, and written to the end of the pre-processed file.
- Lines with comments are completely removed.

File and module inclusions, on the other hand, are not removed. These are just examples, a complete evaluation of how the pre-processor handles different statements should be done.

Pre-processor directives and commented lines should be replaced with empty lines. It is essential not to remove lines or changing the structure of the file when pre-processing since it will make a comparison with the original file difficult.

The suggested modification involves either changing the pre-processor output format, or implementing an alternative output format. If that is done it will be trivial to give the Visual Studio user a view of description files, since all that has to be done is a line-by-line comparison between the original and the pre-processed file.

5.2 Flag finder

A very simple tool was constructed, which was valuable when writing this thesis, and it could possibly be useful for some users. It does not make an attempt to trace the configuration dependencies, it just simplifies searching for references to different flags in the platform. The tool has two modes of operation.

5.2.1 Build-database-mode

To speed up searches a database for the references needs to be built. The database is built for a specific flag prefix. If searches of flags with different prefixes are to be made the database will have to be rebuilt. (Building a database for CFG_ENABLE_-flags takes less than a minute).

Input:	A flag prefix The path to a snapshot view
Output:	A file, flags.txt containing: 1. A list of all flags with the given prefix, together with the total number of references. 2. A list of all flags with the given prefix, together with the names of the modules in which references are made. 3 A list of all modules, together with the set of flags with the given prefix that are referenced in the specific modules. A database file which is used for Flag-search-mode.
Syntax:	`java FlagFinder -build CFG_ENABLE_ "c:\..\snapshot"`

5.1.2 Flag-search-mode

Uses the previously built database search for specific flag searches. A search takes less than a second.

Input:	A flag name
Output:	A list of all files that refer to the given flag, together with the number of references per file.
Syntax:	`java FlagFinder CFG_ENABLE_BUZZER_SUPPORT`

The program does not analyze the references syntactically, it just searches for the occurrences of text strings that match the flag, which are not in a comments. This means that the references are both pre-processor and runtime references. The tool can easily be improved to separate these two cases, and also to do simple syntactical checks. However, relying on the uniqueness of the flag names instead of syntactical check has the advantage that it is faster and it will be made evident if someone misuses reserved flag prefixes.

6 Conclusions and recommendations

This chapter will summarize the most important ideas of this thesis and point at future work.

6.1 Variability management

Managing variability is the key issue of this thesis and feature modeling is presented as a modeling approach, which could be used as a complement to the more commonly used software models. This model can be used for at least three different things, if presented in different forms. Firstly, as a part of the software modeling approach. Secondly, as supporting documentation for configuration. And thirdly, as a marketing document for presenting the configurability of the platform to customers.

Independently of whether feature modeling is used or not, describing variability and variation points is important. There exists a document that does this but it should be more comprehensive.

Configuration requirements should also be documented and incorporated into the development process. It seems to be very important to have clearly defined requirements on configuration to be able to plan the evolution of the platform, and to e.g. foresee what needs to be configurable or removable.

6.2 Suggestions for improving the configuration process

Reducing the number of unnecessary dependencies is a central idea in this thesis. The motivation for doing so is to increase traceability and reduce the number of unnecessary rebuilds. Global configuration files creates unnecessary dependencies. This has the effect that the build tools are unable to determine what needs to be rebuilt after a reconfiguration, which forces costly complete rebuilds of the system.

A set of possible solutions for avoiding this is presented and analyzed. The recommended solution involves a modification of SDE for generating module specific configuration files instead of a global configuration file, as it currently does, thereby reducing the amount of unnecessary complete rebuilds.

Minimizing the build time is also very important and the best way of doing so is to build separate parts of the system, and link them together according to the configuration. However, there are many difficulties that one has to overcome before that is possible, and it did not seem achievable to fit an investigation. But, many of the presented ideas, which are very feature oriented, could prove to be beneficial since features are often a driving force behind software development. And increasing the visibility, in modeling and implementation, of these features, could be advantageous.

An approach for improving readability in description files is suggested. The idea is to give the developers a view in Visual Studio of the description files where statements that are excluded by the SDE pre-processor are marked. This would give developers a way of seeing the results of a configuration on the build process, which would simplify the task of finding errors and of understanding the implications of configurations.

It is important to clarify and document the configuration process. A document should exist, which does not only describe the configuration mechanisms but also the complete configuration- and build-process in detail. Documenting the configuration processes is very important since it affects many areas of software development. And a more common understanding of the build- and configuration-process, and appreciation of associated difficulties, would make organizations more dynamic.

7 Bibliography

[1] A. Deursen, P. Klint
Domain-Specific Language design requires Feature Descriptions
Journal of Computing and Information Technology,
10(1):1–17, 2002.

[2] K. Kang
Feature-Oriented Domain Analysis (FODA) Feasibility Study,
tech. report CMU/SEI-90-TR-21,
Software Eng. Inst., Carnegie Mellon Univ., Pittsburgh, 1990.

[3] Kang, Kyo C. Lee, Jaejoon Donohoe, Patrick
Feature Oriented Product Line Engineering
IEEE Software July/August 2002

[4] Maccari, Alessandro
Industrial Keynote – Feature Modeling in an Industrial Context
PLEES'01: International Workshop on Product Line Engineering – The Early Steps: Planning, Modeling and Managing, Bilbao 2001

[5] Clements, Paul Nothrop, Linda
Software Product Lines, Practices and Patterns
Adison-Wesley 2002 ISBN 0-201-70332-7

[6] L.Griss, Martin Favaro, John d'Alessandro Massimo
Featuring the Reuse-Driven Software Engineering Business
Object magazine, September 1997

[7] Clauss, Matthias
Modeling variability with UML
Diploma thesis

[8] Lorentsen, Louise Tuovinen Antti-Pekka and Xu, Jianli
Modeling of Features and Feature Interaction in Nokia Mobile Phones using Coloured Petri Nets
Springer, Volume:2360/2002

[9] Elsberry, Justin and Elsberry, Nicholas
Using XML and SVG to Generate Dynamic UML Diagrams
Department of Computer Science Central Washington University
http://www.cwu.edu/~gellenbe/docs/xmltouml/xmltechnicalreport.html, March 2003

[10] de Jonge, Merijn
To reuse or to be reused, Chapter 6-7
PhD thesis, Centre of mathematics and computer science (CWI), Amsterdam, 2003

[11] L. Griss, Martin
Product-Line Architectures, (Chapter 22 in) Component-Based Software Engineering: Putting the Pieces Together, May 2001, Addison-Wesley

[12] Schnell, K.; Zalman, N.; & Bhatt, Atul.
Transitioning Domain Analysis: An Industry Experience (CMU/SEI-96-TR-009). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996.

[13] L.Griss. John Favaro and Massimo d'Alessandro
Integrating Feature Modeling with the RSEB.
International Conference on Software Reuse. Victoria, Canada. June 1998.

[14] *Engineering Software Architectures, Processes and Platforms for System-Families*
http://www.esi.es/en/Projects/esaps/esaps.html

[15] Snelting, Gregor
Reengineering of Configurations Based on Mathematical Concept Analysis
Informatik-Bericht Nr. 95-02, January 1995