Master's Thesis

Modeling Dependencies in Dynamic Software Configurations

Vladimir Karadzic D03 & Staffan Thörngren E00

Department of Computer Science
Faculty of Engineering LTH
Lund University, 2007

ISSN 1650-2884
LU-CS-EX: 2007-08

Modeling Dependencies in Dynamic Software Configurations

Vladimir Karadzic & Staffan Thörngren

Department of Computer Science
Lund Institute of Technology
Lund University, 2007

Modeling Dependencies in Dynamic Software Configurations

Abstract

Software development companies are often forced to develop multiple variants of complex software at a high release rate. These circumstances put special requirements on these companies throughout the entire development process, ranging from the design and architecture stage, to the build of the delivered executables. One area that needs special attention when developing a large number of variants, are the dependencies between entities of the software. These dependencies quickly becomes complex. Therefore, considerable efforts often need to be made on acquiring a survey of the dependencies between software components. This is required for the continued development at a constant high rate.

When the software configuration are specified by configuration files that select software components through definition of variables, an overview of the dependencies can be acquired by analyzing these files. In this thesis a tool has been developed to help model and analyze dependencies specified in the configuration files. Since the existing build tool already analyzes the dependencies of the software that are built, the tool should be implemented very tightly with it. The tool should model the parse flow of the build tool which is the starting point of the procedure to discovering dependencies between different parts of the software. This master thesis discusses why such a tool is needed and how it can simplify the software development process. A proof-of-concept prototype was also developed to verify the proposed design.

Table of Contents

1 Introduction **5**

2 Context **6**

2.1 Architecture of the Mobile Phone Software *6*
2.1.1 Features 6
2.1.2 Product Family 7
2.1.3 Variants 7
2.1.4 Module 7
2.1.5 Configuration Files 7
2.1.6 Feature Based Configuration 8
2.1.7 Configuration Variables 8
2.1.8 Configuration Interface 9

2.2 Work Flow *9*
2.2.1 Organization 9
2.2.2 Development Process 9

2.3 Concepts *10*
2.3.1 Granularity of Dependencies 10
2.3.2 Configuration 10
2.3.3 Definition of Static and Dynamic 10
2.3.4 Summary of the Problem 11

3 Analysis **12**

3.1 Core Problems *12*
3.1.1 Configuration Variables 12
3.1.2 Integration Process 13

3.2 Users *13*

3.3 Use Cases *14*
3.3.1 Change Impact Analysis 14
3.3.2 Trace the Configuration Process 15
3.3.3 Comparison of Variants 16
3.3.4 Detection of Unused Configuration Code 16
3.3.5 Visualization of the Configuration 16
3.3.6 Configuration Statistics 17
3.3.7 Reoccurring Configuration Patterns 17
3.3.8 Detection of Logical Errors 17
3.3.9 Configurability of the Source Code 18

3.4 Requirements *18*
3.4.1 Correct Interpretation of the Configuration Directives 18
3.4.2 Model the whole Configuration 19
3.4.3 User Friendly 19
3.4.4 Flexible Design 20

4 Design **21**

4.1 Interpretation of the Configuration Directives *21*
4.1.1 Configuration Files 21
4.1.2 Interpretation Process 21

4.2 Data Structure *22*

4.2.1 Dependencies ... 22
4.2.2 Orthogonality ... 23
4.2.3 Creating the Structure ... 24
4.2.4 Parsing the Structure ... 25

4.3 Usability and Flexibility ... 26

5 Implementation ... 27

5.1 Design of the Prototype ... 27

5.2 Work Procedure ... 30

6 Evaluation ... 32

6.1 Evaluation of the Prototype ... 32

6.2 Related Work ... 34

6.3 Further Work ... 35
6.3.1 Configuration File Architecture ... 35
6.3.2 Support by other Tools ... 36
6.3.3 Analysis of Source Files ... 36
6.3.4 Parse Flow of the Build Tool ... 36

7 Conclusion ... 37

8 References ... 38

Table of Figures

Figure 2-1. The relationship between features, product configuration file, modules and configuration variables 6
Figure 2-2. Feature based configuration. 8
Figure 2-3. Build process. 11
Figure 3-1. One way of showing the impact of a change to the configuration files. 15
Figure 3-2. Examples of logical errors in configuration files. 18
Figure 4-1. The configuration code to the left is transformed to a flowchart shown on the right. 22
Figure 4-2. The flowchart on the right is referencing the configuration code on the left. 23
Figure 4-3. The modified build tool is invoked by a tool to create the model of the parse flow. 24
Figure 4-4. The flowchart is parsed to analyze the dependencies. When interpreting the configuration directives the evaluation logic from the current build tool is used 25
Figure 5-1. A model of the process where the configuration files are parsed by the build tool. 28
Figure 5-2. Content of a process and a decision node 29
Figure 5-3. The process where the model of the parse flow is build. 30
Figure 5-4. The building of the nodes using a modified component of the build tool. 31
Figure 6-1. Parsing of the model 33
Figure 6-2. An example of configuration which were used to verify the prototype 34
Figure 6-3. Configuration 35

1 Introduction

In the past few years mobile phones have become widespread in developing countries. Not too long ago the mobile phone was an exclusive device used primarily by the business elite, while today most people own a mobile phone. This increase does not depend only on an increased need to make phone calls but also by the fact that the mobile phone has, to some extent, been replaced by other devices such as digital cameras and mp3 music players. To keep up the high interest for new phones, the producers have to constantly offer new features. This leads to high release rates where the complexity of the software increases which makes the software development more complicated.

To meet the demands from the market and quickly release new products, Sony Ericsson uses a process which is based on a common design based on product families. This means that an underlying design is created and it is reused by all products in the same family.

Since the products in the family are based on similarity and commonality, even though they are different products, it is possible to release several products relatively fast.

Due to the high release rate and the considerable feature growth, dependencies between software entities are introduced and the software becomes complex. Even though the design goal of the architecture is to make the software components as less dependent on each other, it is not possible to make all components independent. Therefore there is a need for a tool which is able to first spot these dependencies in order to be able to minimize them. The starting point of this procedure is to model dependencies which make it possible to e.g. trace variables or to do change impact analysis. The tool can be used e.g. by people working at the CM department at Sony Ericsson, to model the impact of a change, or software architects to facilitate the structure of software entities.

This master thesis will mainly discuss how a tool can be created for the main goal of is modeling dependencies between software entities. The detailed design of the tool is described and a prototype which is outlined in this report has been implemented.

2 Context

This chapter increases the basic understanding of Sony Ericsson's work flow and processes. The first part of this chapter opens with a description of the architecture of the mobile phone software. This is followed by an overview of the workflow and processes used at Sony Ericsson. In the second half of the chapter some concepts specific to Sony Ericsson are highlighted.

2.1 Architecture of the Mobile Phone Software

Figure 2-1 displays the relationship between features, product configuration file, modules and configuration variables. They are the building stones of the mobile phone software and are among the other concepts described in this section.

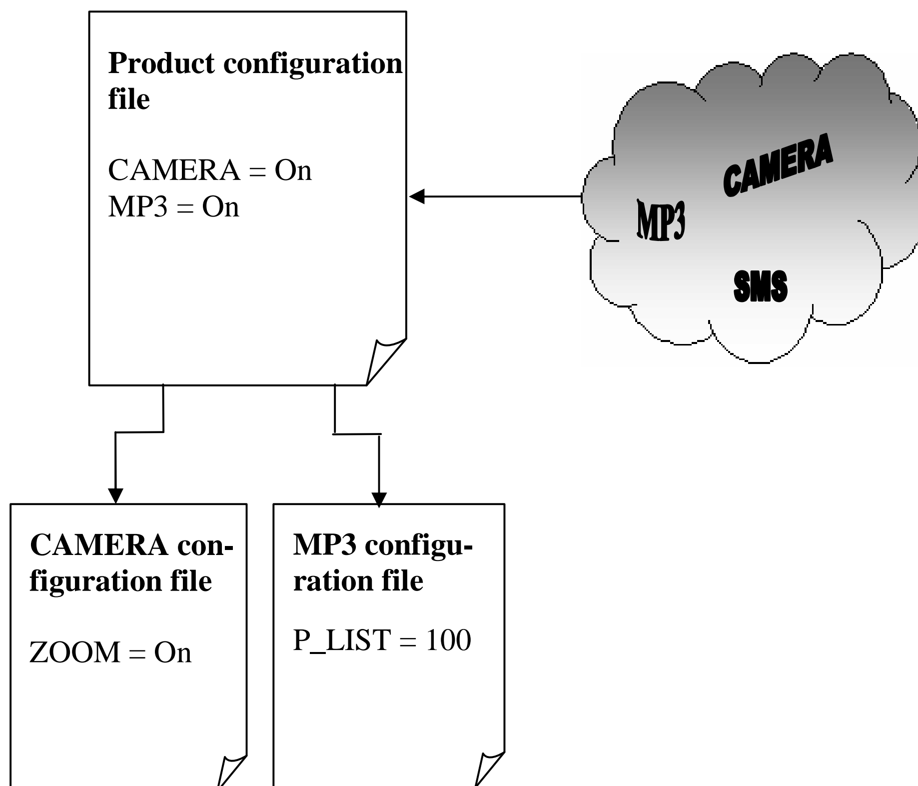

Figure 2-1. The relationship between features, product configuration file, modules and configuration variables

2.1.1 Features

The mobile phone is not only used for making phone calls, it can also be used to play music, web-browse and take pictures. All these functions are called features which together creates complete sets of functionality in the mobile phone. Before the mobile

phone can be produced a description of features is required. When the description is complete, implementation of the features can begin.

2.1.2 Product Family

The purpose of the product family is first to create an underlying design that can be reused on all products in the same family. The product family is based on similarity and commonality which means that different products in the same family contain common or similar software components. This makes it possible to release several different products relatively quickly as well as decreasing cost and time taken to develop the software.

2.1.3 Variants

To meet the existing requirements from the mobile phone market, Sony Ericsson needs to produce a large number of models. Each model must support a large number of requirements. These requirements result in a situation where a very large number of software variants must be developed for each product family.

In this report a variants is defined as a parallel version which has different sub versions created from the common code base.

2.1.4 Module

To avoid the shared data problem, which means that many people simultaneously modify the same code [1], the software functionality is divided into modules. This would enable modules to be separately compiled but also makes complex programs easier to understand.

At Sony Ericsson a group of developers are responsible for one module. The group implements the source code and defines through interfaces which item that are available to other modules. Each module also contains one configuration file that specifies which functionality in the module should be used, for example a certain feature can be enabled or disabled.

2.1.5 Configuration Files

To facilitate the process of integrating the software, Sony Ericsson, uses a process where software parts are selected and configured, from a common code base, according to instructions in text files. These files are called configuration files. Each product family has its own product configuration file where the selection and characteristics of the features that should be included in the final product, are defined. All variants of the mobile phone are also specified in this file.

The product configuration file determines which modules those have to be included in the product. In other words the software parts are encapsulated in modules and configured by a module configuration file referenced in the product configuration file.

The module configuration file specifies what source code is needed for the specific feature. The configuration files are hierarchically organized in several levels of modules and sub modules with the product configuration file at the top.

2.1.6 Feature Based Configuration

Feature based configuration means that features are not dependent on each other which means that they are independent components. They can be combined in different ways and the result will always be a software that meats the requirements. Sony Ericsson supports of this method of setting up the configuration and therefore constantly works with modularization of the software components.

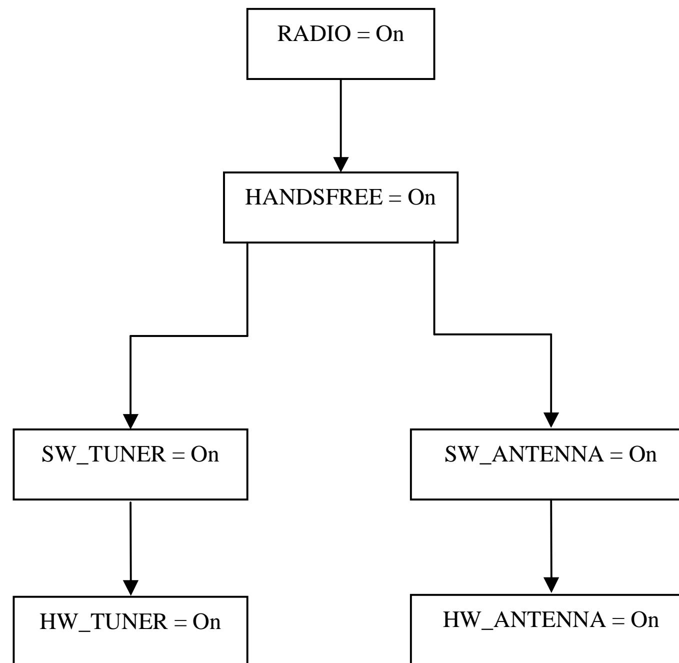

Figure 2-2. Feature based configuration.

Figure 2-2 shows the umbrella model where the feature Radio is enabled in a product configuration file. To enable the Radio feature it is enough to trigger all needed components and functionality which the Radio needs to work correctly. In other words Radio components are encapsulated and completely independent of the other components that build the mobile phone. This leads to simple configuration of products where the developer can easily choose features from a container to set up and form the software in the mobile phone.

2.1.7 Configuration Variables

The purpose with the configuration variables is to define functionality in the software. They can take the form of number, string or array. It should be noted that they have to be declared first. Depending on how the variable is declared there are rules that decide in which scope the variable is readable or writeable.

In other words the variable can be readable by all module configuration files including the product configuration file or only the module where it has been declared. The variable can be assigned the value either in the module where it has been declared or in the product configuration file.

2.1.8 Configuration Interface

The definition of configuration interface is an abstraction of the module to the outside through the configuration file and configuration variables which are declared in the configuration file. This makes it possible to communicate both among modules but also between product configuration file and modules. The purpose is to see, for example which module settings are active or which variables have to be set by the product configuration file.

2.2 Work Flow

This section describes the organization and work flow at Sony Ericsson from a software development perspective.

2.2.1 Organization

The software development is a complicated process which comprises definition and analysis of the problem, implementation and finally verification that confirms that the solution meets the demands. All these characteristics have to be functional, maintainable, testable, easy to use etc, if the software shall be stable and accepted. The time constraints, resources and human error are factors which negatively influence the software development. Therefore the organization both at small and large companies needs to be well structured and defined. The work tasks have to be clearly defined so the employees know what they have to do and who is responsible for specific parts in the company.
Sony Ericsson have geographically distributed development which means that team members are not located in the same geographic region, instead they are in different countries around the world. These cultural differences can be a subtle hindrance to development due to different development styles.

Therefore it is very important that infrastructure and communication between the different sites work. The sites have to be synchronized with each other in turn leads to the efficient work.

2.2.2 Development Process

The first phase of development process at Sony Ericsson is when then function group made the change in the software. When the change is made and developer is satisfied with the modification, the software is delivered to the function test group to be verified. If all tests pass CM integrates the software with other function group's deliveries. If the integration process succeeds successful the software will be tested again to verify that all functions in the mobile phone work.

Each function group is responsible for its module configuration file and configuration variables which are declared there. The configuration management group's responsibility is to first set up the product configuration file and constantly updates it with new deliver-

ies from the function groups. The integration process starts when CM has implemented the delivery.

2.3 Concepts

This chapter defines concepts such as dependencies; static, dynamic and configuration. Then these concepts are put into the context of the environment at Sony Ericsson.

2.3.1 Granularity of Dependencies

The dependencies between entities in the software can be regarded at different levels of granularity. At the lowest level a dependency can exist between two configuration variables. At a higher level modules can be dependent on each other, and at even higher level the dependency can exist between features and functionality areas. Between variants, from the same product family, there exist indirect dependencies since the products share the same code base.

2.3.2 Configuration

The definition of the configuration in the context of this report is the arrangement which decides how the software has to be set up so that it meets the requirements. For instance, a very basic configuration of features for a mobile phone consists of ring tones, phone book, SMS etc.

To build a specific variant, first of all it has to be configured. This means that various configuration variables have to be set and configuration variables values defined. When the configuration is made the integration process starts by parsing configuration files to finally create the executable file.

2.3.3 Definition of Static and Dynamic

In general, the definition of the word dynamic is energetic, capable of action or change, while static means fixed or stationary. In computer context dynamic usually means changeable while static means fixed.

The configuration has to be dynamic to the highest possible degree. This allows changes which mean if initial values of configuration variables are changed the output should be different. If the output is static after each change that means that configuration is incorrect.

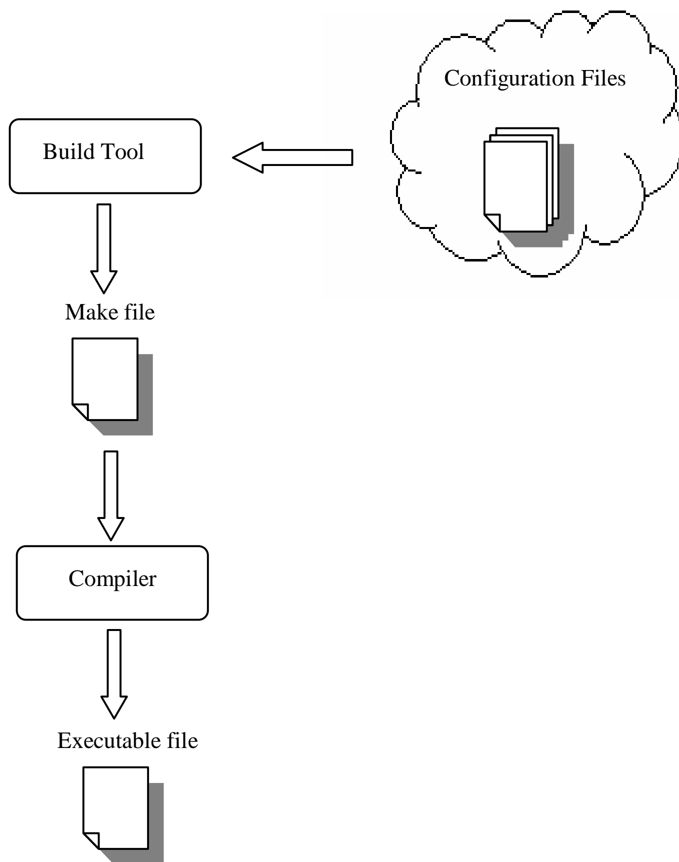

Figure 2-3. Build process.

Figure 2-3 illustrates the build process and defines what part of the configuration is static and which is dynamic. Configuration files are dynamic because they are accessible for changes. In other words, depending on which features that are chosen in the configuration files, the specific variant with these features will be built.

The generated executable file is defined as static. It contains the static defined configuration which is only characteristic for the built variant.

2.3.4 Summary of the Problem

As we have seen the software development is a complicated process. Sony Ericsson has to quickly produce new phones to satisfy demands from the customers. This leads to the fact that functionality in mobile phones increases constantly and consequently the software becomes more complex. The result is that dependencies between software entities increase and therefore there is a need for a tool which is able to first spot dependencies in order to be able to minimize them.

3 Analysis

The previous chapter described the problem domain and some key concepts important to the remaining discussions in this report. This chapter focuses on the analysis of the problem domain. A set of core problems will be defined, followed by a set of possible users of a tool. Then, a set of use cases will be put together. At the end of the chapter the use cases will be transformed to a set of tool requirements.

3.1 Core Problems

The core of the problem domain and its origins will be analyzed in this section. This will be the starting point when the use cases are defined later on.

3.1.1 Configuration Variables

When a software manufacturer is forced to produce many variants of the software at a high release rate, a common solution is to reuse components from a common code base. It is also the approach Sony Ericsson uses. A set of configuration files are used to configure the various variants of a product family. But when the number of variants grows large, while the complexity of the software is steadily increasing, the number of configuration variables gets very large.

To make the source code reusable and to facilitate parallel development, the software at Sony Ericsson is modularized. This is a convenient way to build the architecture to smooth the progress of software development in general. However, during the process of setting up a product, where a set of features is defined, it is not intuitive to make the selection from software modules. During product set up it is more intuitive to use a feature-based configuration process. But the feature-based configuration requires an interface that translates the desired feature to a set of modules. This task is performed by the configuration files in combination with the integration process. Sony Ericsson is aiming to make the product set up feature-based.

Because the act as an interface between the set of features and the software architecture, the configuration files often holds complex dependencies between their configuration variables. And even though not all parts of the configuration are strictly feature-based, the task of translating a feature to the correct set of modules must still be performed by the configuration files. I.e., if a configuration variable representing a certain feature doesn't exist, a set of configuration variables, together representing the same feature, must be set. In the latter case the features are configured by configuration code segments rather than by configuration variables. This scenario results in lengthy configuration files that are hard to read, and even harder to maintain and to trouble shoot.

Another factor that adds to the difficulty to maintain an overview of the configuration files is that module interfaces are too fine-grained. If the set of configuration variables exported by a module controls the characteristics of the module at a too high level of detail, it is often hard to configure this module without detailed knowledge about it. Consequently, configuration variables like these are running the risk of being used improperly, if the software developer does not fully understand the purpose of the variables.

To summarize, the properties of the configuration variables they are:

- large in number,
- dependent on each other in a complex manner, and
- not always easy to comprehend the meaning of.

Consequently it is often hard to maintain an overview of the configuration files, and especially to be able to foresee the impacts of configuration variables changes. This motivates the development of a tool that facilitates tracking of dependencies that can be used when creating configuration files as well as allowing changes to be made in the configuration files.

The properties listed above are closely related to the architecture of the software as well as the module design. Even though these areas are eminently important to the origins of the dependencies in the configuration, it is out of scope of this project to further analyze them. However a basic understanding of where the dependencies originate was needed to be able to design a tool for tracking the dependencies in the configuration.

3.1.2 Integration Process

The software integration process at Sony Ericsson is a complex process utilizing both commercial and in-house developed tools, that are streamlined to meet the special requirements of the software. The environment is constantly being updated as changes are made to the process. Since several product families are concurrently being developed, new versions of the development and build tools must always be backward compatible.

Although the tools are constantly being updated it is difficult to update the changes with the rapid software evolution. Some support for e.g. tracing the use of configuration variables are implemented in the current tools, but a more extended support for tracking configuration variables and dependencies would be preferable.

It would probably be possible, in theory, to use the existing tools to retrieve most of the information needed for an analysis of the dependencies in the configuration; since the tools must ultimately be aware of the dependencies when the build process is initiated. But since the current tools

- have limited support for dependency tracking, and
- are taking care of a large amount of tasks;

they would be a very blunt tool to use. Every single analysis would be very time consuming, and would not be as swift and flexible to be useful in practice.

3.2 Users

The previous section in this chapter analyzed the problem domain. Before the use cases are discussed in the next section, some ideas of who the users of a dependency tracing tool will be outlined.

Possibly the most obvious user group who would benefit from nearly all of the use cases described above, are the software developers. In particular those working with entities of the software that have dependencies to other parts of the software. As these users spend a majority of their daily work working directly with the configuration files, the use cases have been designed with the software integrators in mind.

Other potential users are software architects and designers that could use much of the information provided by the tool to evaluate the intended design of the different parts of the software. Project managers would also benefit from having an overview of the dependencies when crucial decisions need to be made. The information could help to improve prioritizations.

Also people involved in the process of planning the development projects, i.e. in the dialog with the customers, would be better off if the impacts of different changes and modifications were known.

3.3 Use Cases

The core problems described in the previous section contributes to the difficulty of maintaining an overview of the software configuration and of tracking dependencies. The following section focuses on a set of use cases, which are designed to emphasize tasks that are difficult or even impossible to perform in the current environment, as well as other tasks we thought would be useful in general. The use cases will form the foundation for the requirements defined in the next section.

3.3.1 Change Impact Analysis

For software developers the day-to-day work would be simplified if change impact analysis could be done in a swift and flexible way. Making changes to the configuration files would be less erroneous if it would be possible to instantly get an analysis of the impact of the change. This would also speed up the troubleshooting of problems such as faulty configurations. Also software architectures would benefit from having easy access to a change impact analysis.

This use case has many variations. The user starts by making an arbitrary change in the configuration file. Examples of types of changes are:

- turning a feature on or off,
- changing a conditional statement, and
- updating a module's version.

A change could also be a set of different modifications in one or multiple configuration files.

When the change is done the impact analysis could be presented to the user in different ways, at different levels of granularity. Examples of how an impact could be presented are:

- which variants are affected by the change, i.e. what variants use the variable or is otherwise affected by the side and ripple effects of the change;
- at a lower level the impact could be shown as a set of affected modules;
- at an even lower level the impact can be represented by a set of affected source code files;
- at another level the parts of the configuration files that are affected could form the impact;
- the impact could also be presented as a set of configuration variables that are affected by the change, either directly or indirectly by side or ripple effects.

The type of change as well as how the impact is represented differs according to who the user is, and what the purpose of the change impact analysis is. Nevertheless the tool should implement the variations of this use case.

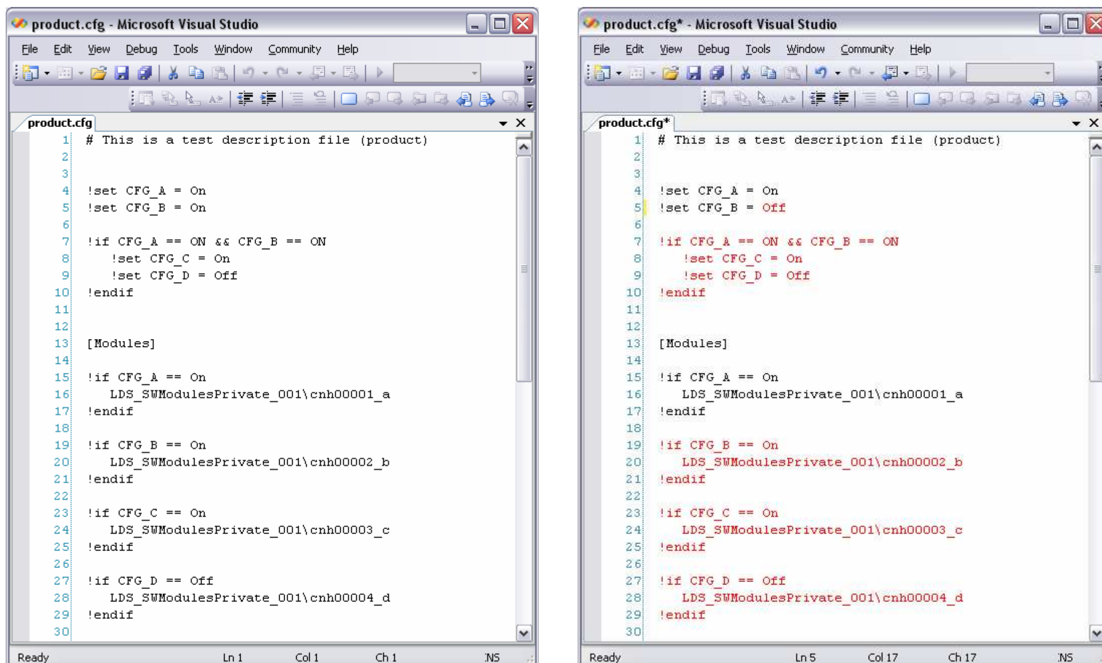

Figure 3-1. One way of showing the impact of a change to the configuration files.

An example of how the impact of changing the value of a configuration variable could be presented to the user is shown in Figure 3-1.

3.3.2 Trace the Configuration Process

The tool should be able to trace the configuration process to increase the understanding of how a certain feature is configured, or what components are dependent on a certain configuration variable. If the execution path of the integration process could be traced and presented to the user preferably in a graphical way, it would be a lot easier to comprehend the relations between the entities of the configuration.

3.3.3 Comparison of Variants

From architecture's perspective an analysis of the differences in configurations between variants in a product family might be of interest. This use case can be regarded as another variation of the previous use case; change impact analysis, where the change is represented by changing the input to the integration process, as opposed to changing the configuration files. This being so, the result presented to the user can be on any of the aforementioned levels; module, source code, or configuration variables.

3.3.4 Detection of Unused Configuration Code

Unused configuration code is configuration code that is not used by any configuration of any variant in a product family. It could be remnants from previous configurations, or simply be part of a larger code segment that is copied from another configuration. When the transparency of the configuration decreases the risk that unused configuration code segments are used increases.

Therefore the tool should be able to detect these unused code segments. The user should be able to analyze a product family and get back a list of the segments of the configuration code that is not used be any configuration. If these code segments are removed the configuration files can be kept as simple as the software permits, thus increasing the transparency of the configuration.

This use case can be regarded as a variation of the previous use case, where all the inputs to the integration process are given all its possible values, and the result is given as a set of configuration code segments. Each set of inputs will give a set of affected configuration code segments. The union of all the configuration code segments represents the used configuration code segments, whereas the inverse to this union, or complement, represents the unused configuration code segments.

3.3.5 Visualization of the Configuration

The purpose of visualizing the configuration is to increase the overview of it. The tool should be able to produce an image illustrating the dependencies to make it easier for the user to gain an understanding of the configuration. Another view would illustrate a flow chart of how the configuration files are parsed by the build tool, to further simplify the understanding of the configuration. The time to get familiarized with a certain configuration would be reduced if the user is able to see all the dependencies in an image.

The level of granularity should be selectable for the visualization as well as for all the previous use cases.

Even though a static image illustrating the dependencies in a configuration is an excellent help when working with the configuration files, an even better aid would be an interactive image. If the user is able to interact with the image and make configuration changes directly in the image, that would be a very intuitive and agile way of working with the configuration.

3.3.6 Configuration Statistics

Statistics is also an important means of acquiring valuable information about a configuration. For instance, a software developer would benefit from knowing how many other modules are using a certain module's interface to be able to make a well-founded decision about a particular change. The information would be possible to gather through using the previous use cases, but a better way to gather the statistics would be to continuously run the tool in batch mode to collect this data. By doing so the users would have instant access to information valuable when doing change impact analysis.

Another great benefit earned by having access to statistics, is the ability to evaluate the module's interfaces, and the utilization of these. If e.g. a decision is made to simplify a module's interface, in an effort to increase the transparency, through decreasing the number of exported configuration variables, having this information at hand would make it possible to come to good conclusions regarding restructuring.

Therefore the tool should be able to gather statistics about the configuration, at different levels. The data should be outputted in a way that facilitates collecting it in e.g. a database.

3.3.7 Recurring Configuration Patterns

A consequence of not having the entire configuration truly feature-based is that similar configuration code segments often occur at different places in the configuration. Since a certain feature almost always needs a set of settings this phenomenon is inevitable without an interface translating the feature to software entities.

There can be different types reoccurring patterns. One type is a set of configuration code lines that is reoccurring. Even though the lines should not be in the same order the tool should be able to detect the pattern. Another pattern occurs in conditional statements where the condition is actually a combination of several conditions. In this case the tool should be able to detect these patterns as well. If a certain set of configuration variables are often evaluated in combination, but seldom individually, it would indicate that the set of variables could be replaced by a single configuration variable.

In order to simplify and tidy up the configuration files the detection of reoccurring configuration patterns plays an important role.

3.3.8 Detection of Logical Errors

Another field of application for a tool is the detection of logical errors in the configuration. By logical errors we mean combinations of conditions that cannot be fulfilled. The output from this use case will be a subset of the output form the use case where unused configuration code is detected. But in this use case another approach is taken to identify the unused code.

Two examples of logical errors are given in Figure 3-2. The left example shows a simple case where the error can be detected by just analyzing the variable names. If the first condition has been fulfilled the tool knows that the second condition can never be fulfilled

just by comparing the current condition with the previous. Accordingly, the second statement can never be fulfilled and the assignment on the third line will never be executed.

```
01 if !A
02   if A
03     B = 2
04   endif
05 endif
```

```
01 if MP3_PLAYER == Off
02   if PLAYLIST == On
03     PLAYLIST_ITEMS = 20
04   endif
05 endif
```

Figure 3-2. Examples of logical errors in configuration files.

The example on the right in Figure 3-2 needs some background information. Suppose that the playlist functionality is turned on if and only if the MP3 player feature is turned on. Then the second condition would never be evaluated as true and the statement on the third line would be unused for all configurations. If the tool had access to this context information even this type of logical error could be detected.

3.3.9 Configurability of the Source Code

If the tool could analyze, not only the configuration files but also the source code, it would be possible to show the valid values for an arbitrary configuration variable, the configurability of the source code. Accordingly this only applies to those configuration variables that are transferred to the source code. The use case will be explained by an example.

Suppose the source code provides the ability to set the screen width to 240 or 128 pixels. The screen width is configurable through setting a configuration variable to an arbitrary integer. But if the variable should be set to any value other than 240 or 128 the build would most likely fail, since the source code wouldn't recognize this value.

If the tool is able to search for this configurability for an arbitrary configuration variable, defined by the user, it would be less erroneous to edit such variables.

3.4 Requirements

In the previous section the use cases was presented. In this section the use cases will be narrowed down to a set of requirements on a tool.

3.4.1 Correct Interpretation of the Configuration Directives

For most of the use cases the tool needs to parse and interpret the configuration in conformity with the current build process. Otherwise the tool's analysis wouldn't be of much use. This being so, the tool should be able to parse and interpret the same set of configuration files, with the same result as the current build process.

This requirement can actually be divided in several sub requirements. The tool must reflect the current build process regarding;

- the set of configuration files,
- the evaluation logic, and
- the parsing algorithm.

If all these requirements are fulfilled the tool is able to perform an accurate interpretation of the configuration directives. This is very important since a faulty interpretation, i.e. an interpretation that does not match the current process's interpretation, will not help a user in the quest of e.g. foreseeing impacts of a change to the configuration files. This is the most fundamental requirement.

3.4.2 Model the whole Configuration

The corner stone of the majority of the use cases is the ability to trace dependencies between entities of the configuration. This is the core of the tool's functionality. But the tool should not only be able to trace the dependencies in a single variant of a product family, but also dependencies through different variants. This requires that the tool needs a model of the whole configuration, as described by the configuration files, as opposed to a certain variant.

To be able to trace the control flow of the integration process the model also has to hold information about the current build tool's parsing pattern. The tool needs not only know what the final values of the configuration variables are, but also how the variables are accessed during the complete integration process.

Accordingly, to meet the requirements implied by the use cases the tool has to model:

- the whole configuration, as well as
- the complete integration process.

3.4.3 User Friendly

The user needs to have easy access to the tool. It should not take more than a few button clicks or a simple command line to start any of the tasks that the tool is designed to perform. For the tool to be used in practice the overhead of learning to use it should be as small as possible. The response time of the tasks should also be as small as possible. The faster the result could be presented to the user the better it is, and the more likely it is that the tool will actually simplify the daily work of its users.

As discussed previously in section 3.1.2, most of the tasks are probably atainable by using the current tools, but they would then never be flexible and easy to perform, and thus never done.

For the tool to be user friendly it has to:

- be integrated in the current work flow,
- present the result as quickly as possible, and
- be interactive.

The second bullet puts special requirements on the tool due to the number of files that needs to be processed, and also the fact that the files reside on network shares.

3.4.4 Flexible Design

As partly indicated by the use cases above, the information the tool has to collect can be used in many different circumstances. Therefore the design of the tool should provide means for other utilities to use the gathered information.

A few of the tasks described by the use cases would benefit from being scriptable. As already mentioned the gathering of statistics from the configuration files is an example of a task that could be scriptable. If this information is collected repeatedly over a time span a measurement of the evolution of the software is provided.

We regard the design of the tool as flexible if the tool;

- provides means for other utilities to use the gathered information, and
- is scriptable.

4 Design

In this chapter the focus will be put on the design of a tool; dealing primarily with analyzing the dependencies between entities in the software configuration. The discussion will be derived from the use cases and requirements described in the previous chapter. At first, the process of interpreting the configuration files will be addressed. The design of the data structure used by a tool will also be dealt with. Finally, some considerations regarding the flexibility of a tool will be discussed.

4.1 Interpretation of the Configuration Directives

To be able to analyze the impact of a configuration variable change, a tool must possess the ability to interpret the configuration directives defined by the configuration files. The process of interpreting the configuration files will be discussed in this section.

In the process of building the executables, the software configuration is basically determined by the configuration files, and the interpretation of these. I.e., the set of configuration files serves as a recipe of what to include in the build, while the set of tools used in the process interprets the recipe to generate the intended artifact. Thus, to make the different kinds of analyzes of the configuration as accurate as possible, a tool must use a set of configuration files and an interpretation process that resemble the original files and process as much as possible.

4.1.1 Configuration Files

Regarding the set of configuration files, no reasons were found not to use the original configuration files as input to a tool. Since the configuration files serve as the definition of the software configuration, and are ultimately plain text files, using anything else but the configuration files themselves seemed pointless. Even though it would be possible to create an altered representation of the configuration files, which perhaps would be easier to interpret, the original set of files would still have to be parsed each time a change was made to them. This would render the extra step of creating a different representation of the configuration files unnecessary.

4.1.2 Interpretation Process

When it comes to the interpretation process, a need to model this procedure was found. As mentioned earlier, the current build tool performs a lot of different tasks and is not streamlined to make this kind of dependency analysis, meaning that it is not the optimal tool to use in this purpose. Accordingly, an altered interpretation process is needed to make a tool perform as desired. However, altering the interpretation process will almost always lower the accuracy of the tool. A small amount of discrepancy might be acceptable but if it grows too large the tool would be unusable. Further reasons to alter the interpretation process are accounted for in section 4.2.1 below.

The discrepancy between the outcome of the interpretation by a tool and the interpretation by the actual integration process is affected by several factors. If the parse flow, i.e. the order in which the configuration files are interpreted as well as how each file is inter-

preted, is altered the result of the interpretation might also change. Therefore, care must be taken when modeling the parse flow not to render the discrepancy too large.

Part of the parse process used when evaluating conditions found in the configuration files is the evaluation logic. The evaluation logic used by a tool should be the same as the one used by the current integration process. Since the evaluation logic is actually defined by the implementation of the build tool the best alternative would be to reuse the same implementation. This would also make the maintenance of the software easier since the double maintenance problem would be avoided.

Accordingly, to meet the requirements defined in the previous chapter, a tool should reuse the already implemented functionality of: acquiring the set of configuration files, and evaluating conditional statements; while the parse flow needs to be modeled in a slightly altered manner to suite the objectives of the tool.

4.2 Data Structure

This section will discuss the data structure a tool needs to be able to perform the previously defined set of tasks.

4.2.1 Dependencies

The main objective of the design is to model the dependencies between entities in the software configuration. With the information about the dependencies at hand the use cases described earlier would be possible to implement. The only possible way to model the process, that satisfies the requirements described earlier, was found to be through a flowchart. Hellström and Pileryd arrived at the same conclusion in [2].

If the conditions used in the configuration files are represented as decision nodes, and the statements in between the conditions are encapsulated in process nodes, a flowchart like the one in Figure 4-1 could be created.

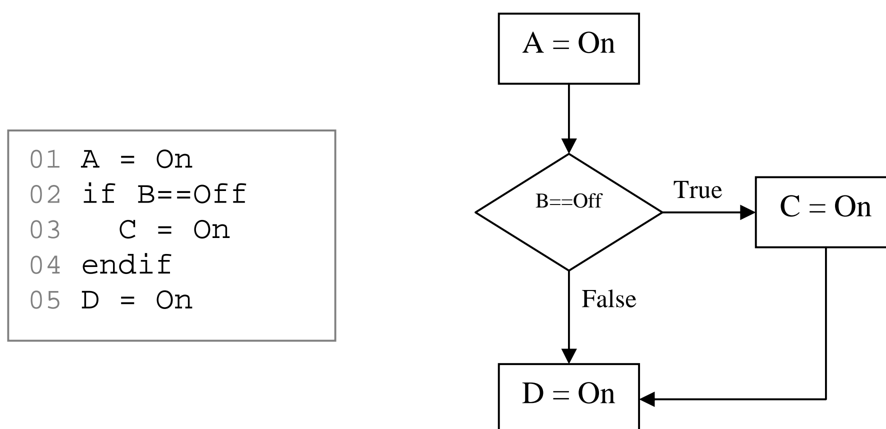

Figure 4-1. The configuration code to the left is transformed to a flowchart shown on the right.

When the complete set of configuration files are transformed into a flowchart, the foundation of the dependency analysis is created. Parsing the flowchart, evaluating the conditions in the decision nodes, will make it possible to gather information needed to analyze the dependencies. Hence, a tool needs to parse the configuration files according to the flowchart instead of imitating the normal parse flow of the build tool.

4.2.2 Orthogonality

In the example above, the contents of the configuration files are combined with the information about the parse flow of the files. This will make it harder to update the flowchart after a change has been made to one of the configuration files. But if the contents and the parse flow of the configuration files could be separated, the same flowchart could be used even after a change of the configuration was made, and the tool would be more flexible to use.

To make the contents and the parse flow of the configuration files orthogonal; instead of embedding the configuration code in the flowchart, references could be used, as in Figure 4-2. However, referencing the configuration directives by line number doesn't make the contents and the parse flow orthogonal, since changing the configuration files in a manner that the lines are moved, would require a regeneration of the flowchart. As long as plain text files are used to hold the configuration directives, it is hard to eliminate the correlation without using some sort of tags marking where the nodes are.

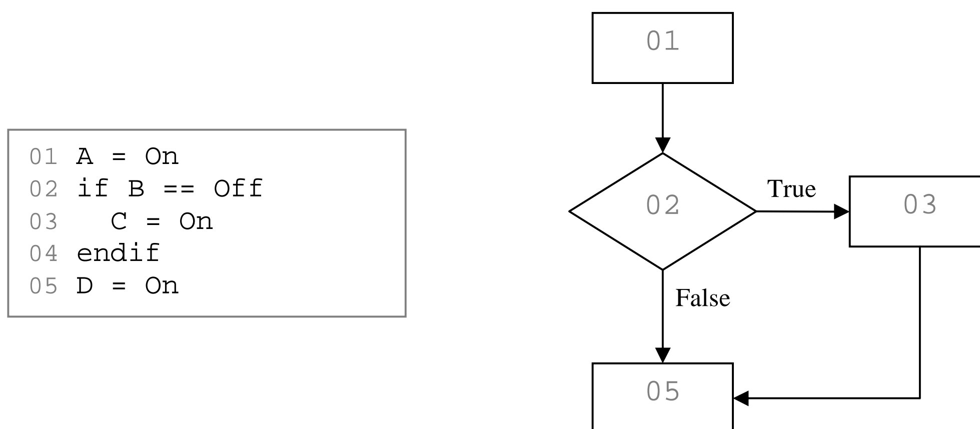

Figure 4-2. The flowchart on the right is referencing the configuration code on the left.

Since changing the way the configuration directives are represented was out of scope of this project, line numbers were chosen to constitute the references in the flowchart. When the configuration consists of more than one file the file name also needs to be a part of the reference.

While simple changes like changing the value of a variable assignment never require a rebuild of the flowchart; making the contents of the configuration files and the model of the parse flow totally orthogonal is not possible. Since the flowchart is modeling the pars-

ing of the configuration files, changing the code in such a way that a new node should be created in the flowchart, always require a rebuild of the flowchart. But, on the other hand, if code segments are removed from the configuration files, the same flowchart could be used if the nodes that should be removed are regarded as empty nodes. Accordingly, different types of changes require different types of actions in order to keep the flowchart up-to-date.

4.2.3 Creating the Structure

To build the flowchart the complete set of configuration files must be parsed. One way to do this is by building a parser, which goes through the files. But since the current build tool already possesses the ability to acquire and parse all the configuration files, it appeared to be a better solution to integrate the functionality to build the flowchart in the current tool.

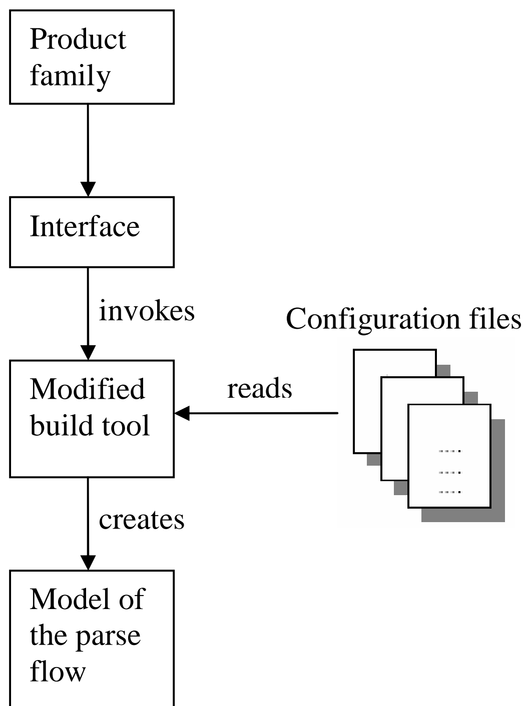

Figure 4-3. The modified build tool is invoked by a tool to create the model of the parse flow.

Hence, the build tool should include a feature to build the flowchart. This would facilitate the creation of the required data structure by invoking the current build tool with the purpose of building the flowchart. The process is shown in Figure 4-3. The creation of the flowchart should create a representation of the complete product family. This is done by invoking the current build tool in a manner that it parses the complete set of configuration files.

4.2.4 Parsing the Structure

When the data structure has been created the dependencies can be analyzed by parsing the flowchart. Again, the best way to do this was found to be by reusing as much of the current build tool as possible. Accordingly, if needed, the modified build tool should be invoked when the flowchart is parsed. For a tool to be able to perform the use cases it has to gather information about what parts of the configuration files are being parsed, which configuration variables are accessed and what source files are to be included.

During the process of parsing the configuration directives pointed out by the nodes in the flowchart have to be interpreted. This is optimally achieved by invoking the evaluation logic from the current build tool. The process of parsing the structure and gathering information about the configuration is shown in Figure 4-4.

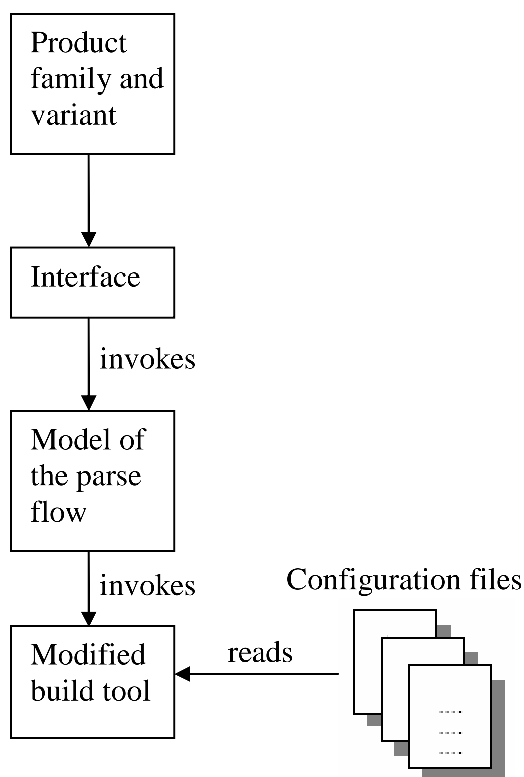

Figure 4-4. The flowchart is parsed to analyze the dependencies. When interpreting the configuration directives the evaluation logic from the current build tool is used.

While going through the flowchart, a tool should acquire the required information such as the parse flow, the accessed configuration variables and the included source files. The data gathered by a tool should be presented to the user at the end of the flowchart analysis.

4.3 Usability and Flexibility

For a tool to be easily integrated in an existing work flow the design has to be user friendly and flexible. This section will account for these considerations.

If the behavior of the other tools used in the integration process is imitated, it would be easy for a user accustomed to the current process to use the tool. Since most of the tools used in the current integration process can be invoked by either a command line or by a command button from within another tool, the same invocation methods should be implemented by this tool.

Providing a non-graphical user interface facilitates the tool to be scriptable. If a tool is giving its output in plain text, in an easy-to-parse format, other tools could be used to further analyze the result.

How the information acquired by a tool could be used by other tools of the integration process is illustrated by a change impact analysis example. When the user has loaded a set of configuration files in the text editor the tool is invoked, by pushing a button, to show what lines in the files are really used as a result of evaluating the conditions. The tool outputs the file names and line numbers that should be highlighted in the text editor. After the user makes changes to the configuration files another button is pushed to highlight the impact of the change. The tool is then invoked to output the difference in what directives of the configuration are used after the change. The tool outputs information about the new set of directives based on the interpretation. Now the text editor is able to highlight the directives that are no longer parsed as well as the directives that have become active due to the modification of the configuration.

Providing the user such a means of visualization would make it much easier and more intuitive to work with the configuration files. If the tool is providing interfaces that can be used by other tools in the integration process, the work of architectures and integrators could be simplified.

5 Implementation

In this chapter the implementation of the prototype tool will be discussed. What is implemented as well as how it is implemented is briefly described. To increase understanding the architecture will be illustrated.

5.1 Design of the Prototype

The prototype can be regarded as a proof of concept where functionality such as impact analysis partly works. All use cases, discussed in this report are not implemented but are described in detail for possible future development.

The core of the prototype's functionality is to trace dependencies between entities of the configuration. The first step of discovering dependencies is to build a model of the parse flow which is described in the configuration files. This makes it possible to parse the complete configuration relatively quickly since there is no need to parse all configuration files for each tracing.

To build the model that contains all variants, all of the configuration files have to be parsed once in order to get a complete configuration reflected in the model. Then the model is created and ready to be reused for each request. In other words when the user for example changes some value on configuration variable to analyze impact of the change, the model will be parsed instead of configuration files. This method saves time since parsing the model takes less time compared to parsing all configuration files.

Figure 5-1 describes how the model of configuration parse flow is created from the configuration files.

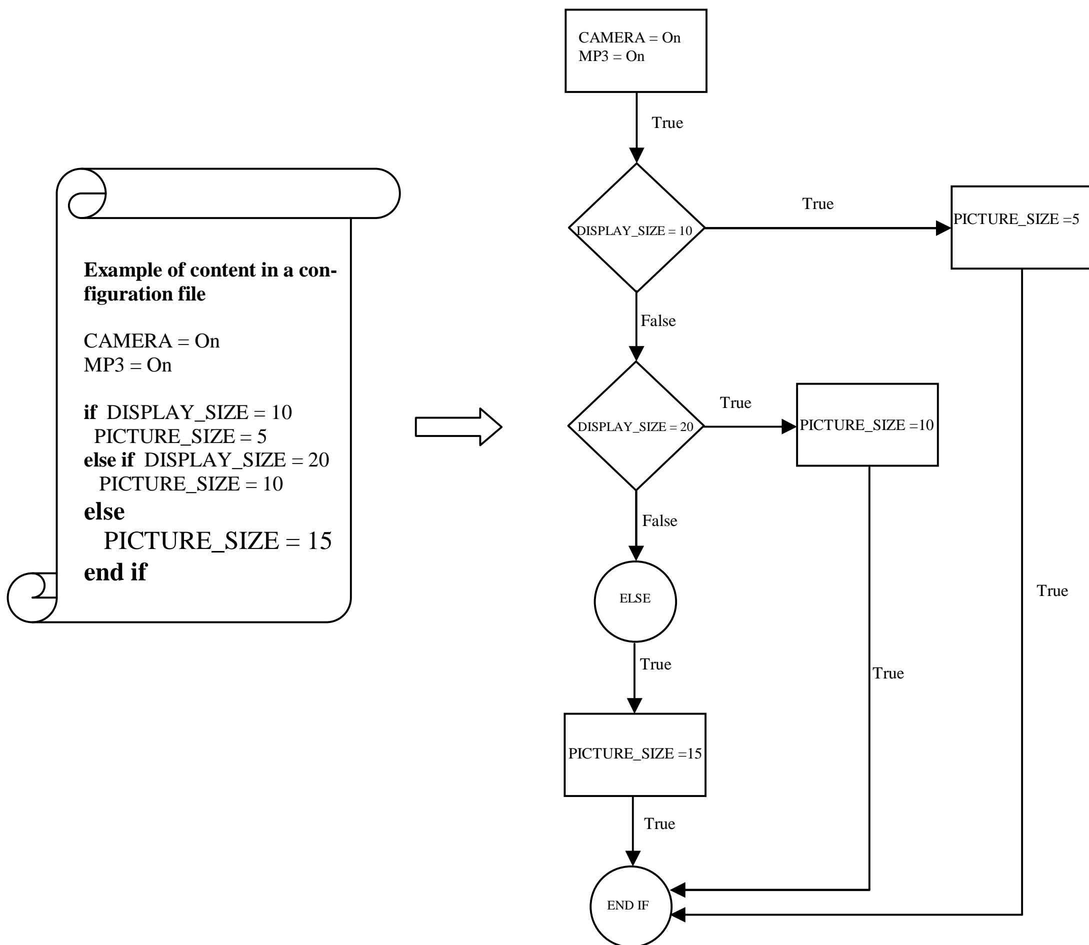

Figure 5-1. A model of the process where the configuration files are parsed by the build tool.

The model contains all information from the configuration and forms a flow chart with logical statements and configuration variables. In Figure 5-1 the flowchart starts with the node which contains only one statement. The arrow from the node has always the true value which means there is only one possible way, to move forward from the node in the flow chart. These nodes are called process nodes. The next node is a decision node that contains the *if* condition and unlike the previous node, it has two arrows, true and false arrows. Depending on if the *if* statement is fulfilled the corresponding arrow is used to refer to the next node. The node with the *else if* statement is also a decision node and has also two arrows similar to the node with the *if* statement. Nodes containing *endif*- or *else*-condition have only one arrow similar to the first node in the flow chart described above. There is no need to have two arrows due to the fact that the *else* or *end if* conditions are always true.

The implementation of the model is integrated in the current build tool for several reasons. The parsing algorithm has already been implemented so it would be completely unnecessary to implement our own algorithm. Another reason is the double maintenance

problem. Each time the syntax in the configuration files is changed it would be necessary to update both parsing algorithms.

The nodes in the model contain all the information that is needed for the tracing. All nodes contain a configuration filename and line numbers which represent the content in the nodes. They also contain the reference to the next node. Decision nodes also contain *if*, *else if* and *else* condition, which are evaluated with functionality in the build tool.

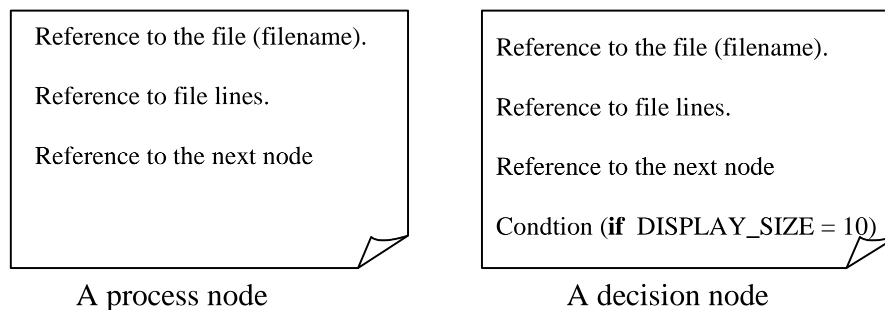

Figure 5-2. Content of a process and a decision node

For example, to see the impact of the change of a configuration variable's value the model will be traversed. Each visited node gives information of which file and which line numbers should be read. With the help of this information the correct reading will be made under condition that modification has not changed line numbers. In cases where a line number has been changed the model has to be rebuilt.

The model does not cover the entire configuration which means that the results from implemented use cases do not give totally completed information due to special cases, but it is still useful as a proof of concept.

Two use cases have been implemented, trace variables and trace path. The first one gives information about the variables that are affected and the second gives information about which path is traversed when the configuration is changed. In both cases the list containing information about which variables that are used and which original value they have or which path has been traversed, has to be stored first to be able to compare with the generated list. The comparison can be made with a tool that is able to compare text files.

To handle the prototype an interface is implemented. The user gives an instruction to the prototype through the command window. The instruction is read by the interface which starts the desired process. Figure 5-3 below illustrates the build process of the model.

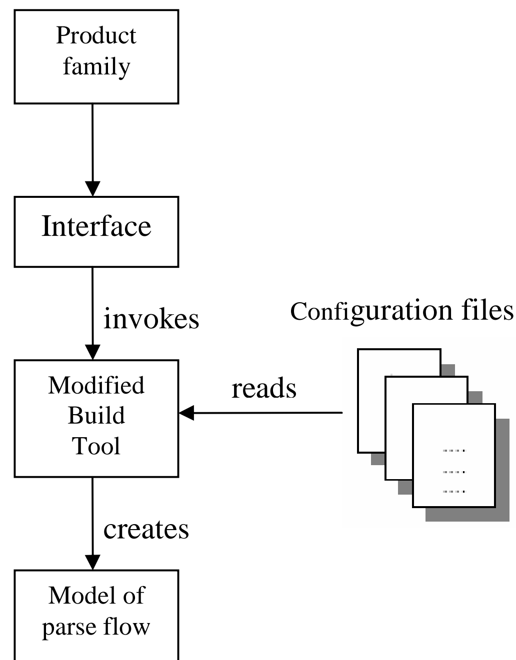

Figure 5-3. The process where the model of the parse flow is build.

5.2 Work Procedure

The first step of the thesis was to gather all necessary information about the integration and build environment at Sony Ericsson. On the basis of the gathered information, the decision on how and where the prototype should be implemented was made. Since the parsing algorithm was already implemented in the build tool, the simplest way to start with implementation was to modify the build tool and make use of the parsing algorithm instead of implementing a new one. This means that the interpretation of configuration files in the prototype is always the same as in the build tool.

The next step was to implement a model of the entire configuration that is described in the configuration files. Since the build tool detects all events which the model has to make use of, for example to build a decision node when build tool detects an *if* statement, the nodes of the model are built in the section of the source file where the detection occurs. In other words, the model is implemented in the build tool and the following Example 1 describes in which sections of the parsing algorithm source file the building of the model's nodes takes place.

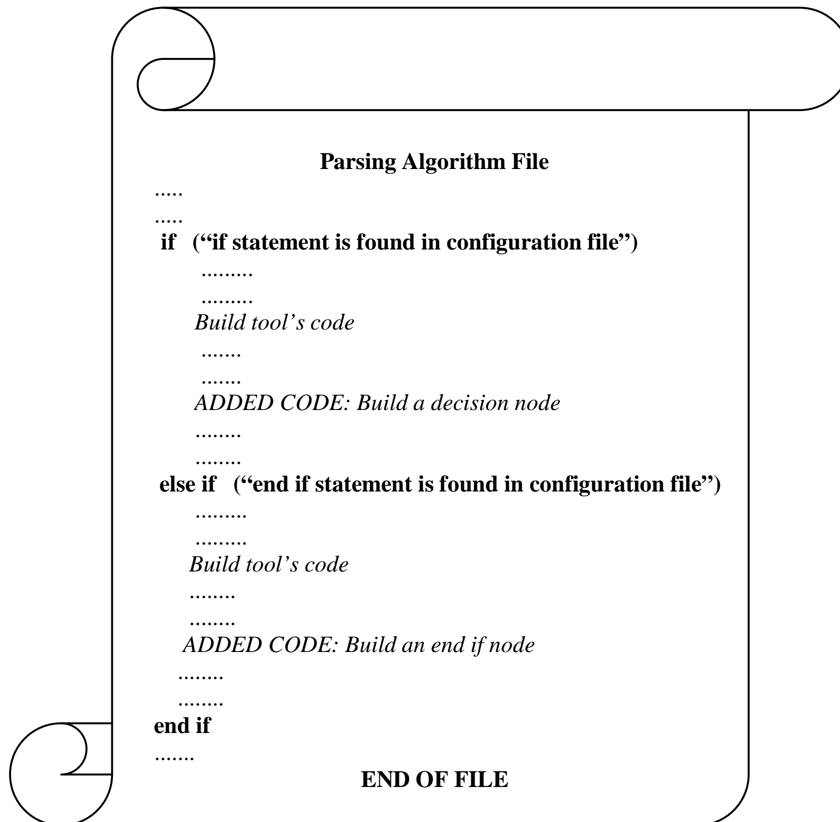

Figure 5-4. The building of the nodes using a modified component of the build tool.

Due to size and complexity of the original configuration files test configuration files are used. In the start these files were constructed very simply, only containing a couple of *if* statements, but in parallel with solving these simple cases the complexity of test files was increased. Each time the case was solved the model was also built on the original configuration files to verify that the implementation was correct.

When the model was finished the implementation of the use cases begun. Due to the fact that the structure of the model took more time than was estimated only two use cases were implemented. The detailed design of use cases was made in the beginning which led to a relatively simple implementation.

To execute a use case it takes the same time for the build tool to parse all configuration files. This is not strange since the model is implemented in the build tool and all configuration files have to be parsed to gather the completed information. The information represents content in the nodes of the model.

6 Evaluation

Along the course of this project many ideas of changes and improvements emerged, which were outside the scope of this project. This chapter will summarize these ideas. The first section evaluates the prototype while the second section discusses amongst other topics the structure of configuration files.

6.1 Evaluation of the Prototype

The main objective of the prototype is to trace dependencies between entities of the configuration. This leads to detection of system problems which makes it possible to isolate problems and finally determine them.

The prototype does not implement all use cases but it is able to partly discover dependencies between configuration variables. It does not give a complete overview but it is useful like proof of concept.

The strength with the prototype is modelling of the parse flow. The model describes all possible paths through the configuration and when this information is available all analysis of discovering dependencies are possible to perform. Therefore, most of the implementation time in this master thesis was spent on implementing this model. Instead of implementing an independent program, implementation is done in the build tool, which complicates the implementation of the model. To understand how the current build tool is implemented and how it works took more time than was estimated from the beginning. It appeared that the implementation part was more complex than planned; due to all the special cases that occur when a build tool parses configuration files. Since the prototype follows the build tool's parsing flow to build the model, it was necessary to understand exactly where in the build tool the interpretation of the configuration files occurs and how it is implemented. It was a strenuous way to go but at the same time the most efficient since in the future it will be easy to implement the complete product.

Building the model takes time which is necessary for the build tool to parse all configuration files. But when the model is built there is no need to parse the configuration files again since the model holds all information about configuration files. Therefore, when the user wants to execute a use case, for example, to see the impact of the change, the model will be parsed. This is more efficient than to parse all configuration files again.

The desired situation is that the configuration flow and configuration files should be totally independent from each other after the build job is complete. In the prototype the model is built while build tool parses configuration files and the parser simultaneously stores logical statements and line numbers from configuration files in the model. When the model is built and the user performs some use case the logical statements which are stored in the model are evaluated by the build tool. Each node contains file and line numbers which should be read when the model is parsed.

To get an independent product the dependencies between the parse flow and configuration files should be determined.

Figure 6-1 describes how the parsing of the model is pursued in the prototype.

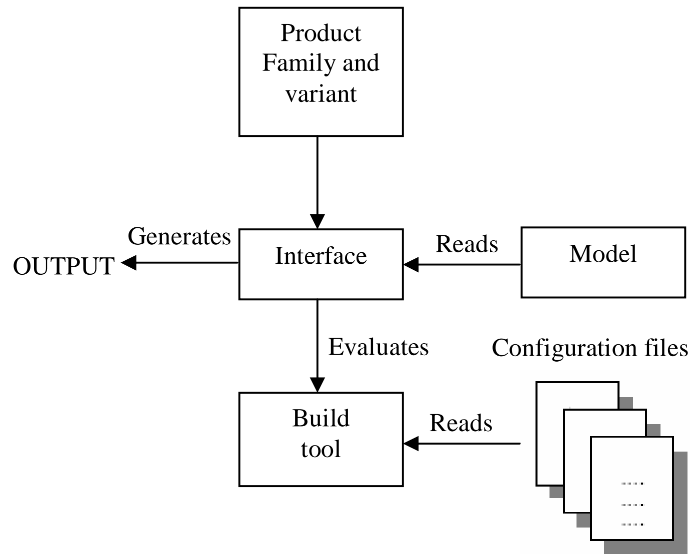

Figure 6-1. Parsing of the model

The graphical user interface is not a part of the prototype. Instructions to the prototype are given through the command window, but it would be much more user friendly to have a graphical interface. But otherwise the target of this prototype is people with good computer knowledge.

6.1.1 Example

To verify that the prototype is modelling dependencies correct the example in Figure 6-2 was used.

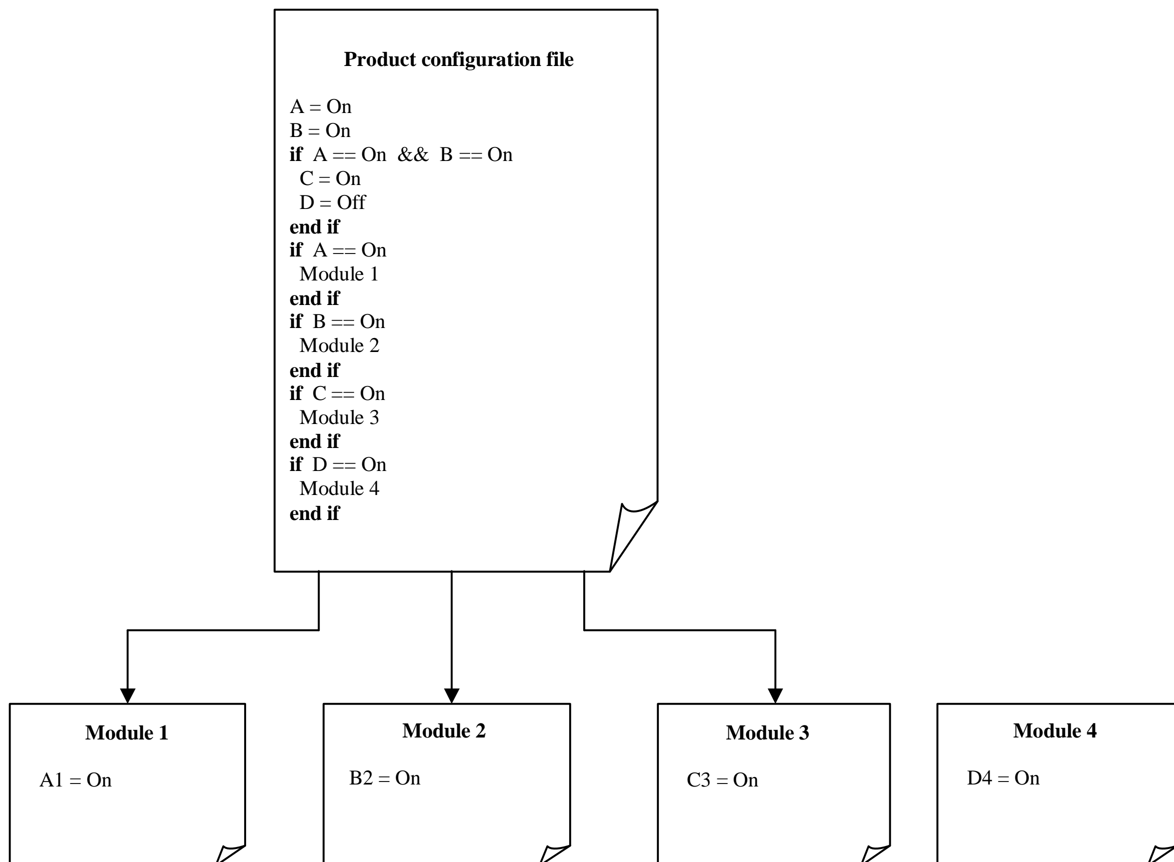

Figure 6-2. An example of configuration which were used to verify the prototype

The product configuration file contains four configuration variables: A, B, C and D. It also references four module configuration files. Each of the module configuration files contains one configuration variable. If the value of configuration variable A changes to Off then the product configuration file will not reference Module 1 or Module 3 and variables A1 and C1 will never be set. Another case occurs if the value of D is changed. Then the reference from the product configuration file will be created and D4 will be set. This example has been used to verify that the prototype behaves correctly.

6.1.2 Risks

The method of how and where the prototype should be implemented was well chosen but the miscalculation was time estimation. The school and industry tasks are very different. The industry tasks are often complicated where unpredictable cases can occur. It is very important to be aware of this factor when estimating time. Otherwise, if we did it all over again the same strategy would be used except time estimation would be different.

6.2 Related Work

The previous master thesis [2], at the CM department at Sony Ericsson was about static dependencies between configuration variables. The authors defined as dependencies like dependencies which do not take the values of configuration variables into account unlike

the dynamic dependencies. In this report the definition of dynamic and static dependencies are different.

```
if DISPLAY_SIZE = 10
   PICTURE_SIZE = 5
   ..........
```

Figure 6-3. Configuration

In Figure 6-3 two configuration variables are given. In a previous master thesis work [2], the authors defined that the dependency between DISPLAY_SIZE and PICTURE_SIZE is static if the value on DISPLAY_SIZE is not taken into account and dynamic if the value is taken into account. In this master thesis we choose not to define dependencies between configuration variables as dynamic and static. Instead the configuration is defined as dynamic and executable files as static (see also 2.3.3). In other words the meaning of static and dynamic is used different in this report where dependencies between configuration variables are defined neither as static or dynamic.

But these two master theses are very similar since the goal is the same;

1. to first investigate the environment at Sony Ericsson and
2. to implement a tool which traces dependencies between entities.

Therefore the previous master thesis was the starting point to make decisions in this project. For example in the previous project the tool which traces dependencies was completely independent from the build tool. This means that a new parsing algorithm was implemented which means when configuration files are changed the program stopped to work. Therefore in this master thesis another decision was made, to implement the program very closely with the build tool. This method means that the double maintenance problem is avoided but it also has some disadvantages. It took a lot of time to understand the parsing algorithm which was not the case in the previous thesis.

How to model dynamic dependencies is also a good advice from the previous master thesis. Authors proposed that the model which contains configuration should be considered like a flow chart instead of a graph. This led to the model being created like a flow chart where the nodes contain the logical statements and configuration variables. These observations were very helpful to ensure that the correct decisions were made in this master thesis work.

6.3 Further Work

This section gives suggestions to work that should be performed to further investigate the dependencies between entities in the software configuration at Sony Ericsson.

6.3.1 Configuration File Architecture

It is recommended that the structure of the configuration files is reviewed. The number of configuration variables exported by the module’s interfaces might be too large. It would

be simpler to set up and maintain product configurations if the number of configuration variables were reduced. The configuration process might also benefit from a more structured design of the configuration files, e.g. an object oriented architecture.

6.3.2 Support by other Tools

If a decision is made to implement a tool like the one proposed in this report at Sony Ericsson, then we recommend that support for interpreting the result outputted by the tool should be implemented in other tools used in the integration process. An example was given in section 4.3 illustrating how a text editor could be used to facilitate the analysis of dependencies.

6.3.3 Analysis of Source Files

This report primarily deals with the analysis of configuration files; but to get a more complete picture of the dependencies an analysis of the source files should also be performed. To be able to implement the configurability use case described in section 3.3.9, source files also have to be parsed. The information should then be added to the information gathered by this tool to provide a more complete description of the dependencies.

The tool described in this project is integrated in the current integration process. Maybe the same approach could be taken to make a similar modification of the build tools to perform a corresponding analysis of the source files.

6.3.4 Parse Flow of the Build Tool

Since a tool used to analyze the configuration most certainly needs to imitate the parse flow of the original build tool used, the manner in which the build tool parses the configuration files has a great impact on the design of the tool. During this project we were not able to make an accurate model of the parse flow, due to the complexity of the build tool. The result is that the tool is not able to give a completely accurate analysis under all conditions.

If the parse flow of the build tool was more straightforward and uncomplicated, for example, if the different configuration files were parsed only once and in an intuitive order, add on tools like the one suggested in this report would easier to develop and probably also lead to more accurate results.

7 Conclusion

Due to the demands and requirements from the market, software manufacturers might be forced to develop a large number of variants of their software, with a high release rate. This puts special demands on the processes and tools used in the development process, as well as on the software itself.

To be able to keep up the rapid software development of multiple variants, it is an absolutely necessary for software manufacturers to have the ability to obtain and maintain a survey of the dependencies between the entities in the software. The knowledge about the dependencies gives valuable information about what parts of the software are affected by a specific change. This type of information could be used to make well-founded decisions regarding the development process by people at different levels of the company, ranging from developers and architectures to project managers, strategists and other decision makers.

The configuration of the software, which controls what components are included in the executable, is determined by the configuration files describing the components, as well as the process of interpreting these files carried out by the build tool. Since the dependencies between entities of the software are ultimately defined by the build tool in use, a tool capable of analyzing the dependencies must imitate this behavior as closely as possible. Accordingly, the best way to implement such a tool appeared to be by reusing parts of the current build tool, e.g. the evaluation logic. However, since the build tool used at Sony Ericsson has reached a very high level of complexity, certain parts of the build process had to be modeled in a simpler manner to be useful for dependency analysis.

The prototype developed during this project shows that it is possible to implement a tool in the proposed manner: reusing components from the build tool, as well as modeling the parse flow of the build tool. This approach of implementation makes it possible to: make an accurate model of the dependencies; integrate the tool more closely with the current development environment; it also makes the tool easier to maintain whenever the build tool is updated. Only basic functionality is implemented in the prototype but the foundation of modeling the dependency is included which is the basis of all other use cases described in the report. Successful tests were carried out; where test configuration files were used to verify that the prototype modeled the dependencies correctly.

Since the ability to model the dependencies is closely related to the build process, it appeared that the design and architecture of the build tool has a considerable impact on the ability to model the dependencies. E.g., if the some configuration files are parsed multiple times the parse flow is harder to model than if each file is parsed only once. Therefore, considerations regarding the possibilities of modeling the dependencies should be made during process of designing the build tool.

We strongly feel that the software development would benefit considerably by having a tool that is able to model the dependencies in software configurations, described by this report.

8 References

[1] Wayne A. Babich, *Software Configuration Management: Coordination for team Productivity*, 1986, ISBN 0-201-10161-0.

[2] Hellström & Pileryd, *Controlling the Variant Explosion*, 2005, ISSN 1650-2884.