Master's Thesis

Controlling the Variant Explosion - Enforcing Stability in Highly Configurable Large Scale Software

Anders Hellström D03 & Björn Pileryd D00

Department of Computer Science
Lund Institute of Technology
Lund University, 2005

ISSN 1650-2884
LU-CS-EX: 2005-26

Controlling the variant explosion

- Enforcing stability in highly configurable large scale software

Abstract

The market for mobile phones is rapidly growing. The product lifetime of a mobile phone is now down to about a year in Europe and just a few months in Japan. The software functionality has become very complex and the operators and the users require that the products can be customized to meet their specific needs. This forces the producers to rapidly create many variants of the products. The software must be highly configurable while on the same time modular and stable. The overall goal of the master thesis work is to enforce the software development environment to meet this challenge. The report consists of three parts. The objective of the first part is to investigate methods to configure software variants in a more dynamic way than it is done today and develop a tool to compose and build software variants. The second is to investigate how to visualize dependencies between the configuration variables extracted from the configuration files which are used to configure the different products, and the third part is to investigate how a framework of rules can be used to validate the configuration files. Beside the actual results the report consists of suggestions of future work that Sony Ericsson can conduct in this area.

Preface

The target audience of this report mainly consists of two groups; Sony Ericsson Mobile Communications employees and persons associated with Lund Institute of Technology, e.g. the thesis tutor, examiner and opponents. The reader is expected to have knowledge about computer science; preferably the reader shall be familiar with concepts such as software configuration management, abstract data types, compiler theory and creation of applications using C++ and Microsoft Foundation Classes.

In the report some abbreviations and terminology are used. These are explained in Appendix A and Appendix B.

Table of contents

1 Introduction 4
2 Dynamic Variant Tool 8
2.1 Analysis 8
2.2 Design 13
2.3 Implementation 15
2.4 Conclusions 15
3 Configuration variable dependencies 18
3.1 Analysis 18
3.2 Result and discussion 22
3.3 Conclusions 26
4 Framework of rules 28
4.1 Analysis 28
4.2 Result and discussion 29
4.3 Conclusions 35
5 General conclusions 36
6 References and recommended literature 38
Appendix A – Abbreviations 39
Appendix B – Terminology 40
Appendix C – Screenshots of DVT 42

1 Introduction

The mobile phone industry has changed rapidly during the past few years. The relation between phone producers and the operators and customers has been a “take it or leave it” situation where the producers would offer phones and the operators would decide if to adopt the product or not.

The number of mobile phone users has increased tremendously and the market is intensified. The product lifetime of a mobile phone is now down to about a year in Europe and just a few months in Japan.

The tightened competition and increased availability of mobile phones has shifted the power to shape the product more towards the operators. Now operators want to offer services that other operators do not have, and the phones are tied closer together with operator services as a whole product package by customizing the phone accordingly. Phone users also require the ability to personalize their phones with e.g. different images and ring tones.

In order to be successful on the mobile phone market Sony Ericsson Mobile Communications, SEMC, must be able to quickly meet the demands and rapidly produce new products with new features.

Core problem description

To meet the different demands from the operators and customers the software in the phone needs to be configurable in many ways. The differences between the demanded functionalities are often relatively small compared to the total amount of source code in the software. To reduce the time and resources needed to produce these different software variants, existing code is reused.

The ideal situation for SEMC when setting up a new mobile phone variant would be to have a shelf of available feature components to choose from to compose the desired functionality. These feature components would all be compatible and no matter how they were combined, they would always produce perfectly working phone software.

The real situation is however different. Because today’s phones offer many applications and services besides phone calls the functionality has become very complex. Features are not totally modular as in the ideal situation. A feature often interacts with other features and expects the existence and a certain behavior of these.

SEMC now faces the problem of keeping the software components as modular as possible and in the same time allowing functionality to interact closely. On top of this the software should be configurable on a detailed level.

SEMC software development domain description

A software project aims to develop software that results in a product, in this case a mobile phone. There can be several variants of each product, e.g. a product released in the Chinese market needs to be adopted to meet the needs of that market. A variant can be described as a variation of the software developed to provide slightly different behavior than the original version. Configuration files are used to define and describe the contents of products. A configuration file contains information about whether or not software functionality shall be enabled or disabled. There are two types of configuration files; the product configuration file and the module configuration file. They both consist of the same type of information but are used slightly different. The product configuration file is used to define and describe an entire product and the module configuration files are used to define and describe the modules. The configuration files also specify which software source code files that are to be used. The configuration management department is responsible for the product configuration files. The modules are used to group functionality and features and consist of software source code. Each module has one configuration file and for each module there is a group of developers that are responsible for it. SEMC uses ClearCase, which is a software configuration management system, to handle all products, variants, modules, source code and configuration files.

A configuration variable is a mechanism in the configuration file to enable or disable functionality in the software. A variable has the same properties as a normal programming variable in terms of available actions, which means that a configuration variable can be declared, undeclared, read or assigned a value. There are three types of configuration variables; public, private and product. Variables that have been declared public can be read by any module. Private declared variables are only accessible in the scope that they have been declared in. The product variables are declared in the modules and assigned values in the product configuration file.

SEMC uses the Software Development Environment, SDE, as their main development and build environment. It provides a unified interface to software development and configuration management tools. SDE also manages the complete build process for software products and software modules. SDE provides a mechanism, an extra configuration file, which is only used in user private builds. This file is called the descrextra. The file can easily be modified to describe files and/or modules that are to be added or removed from the product. It can also be used to create, remove or change the value of configuration variables.

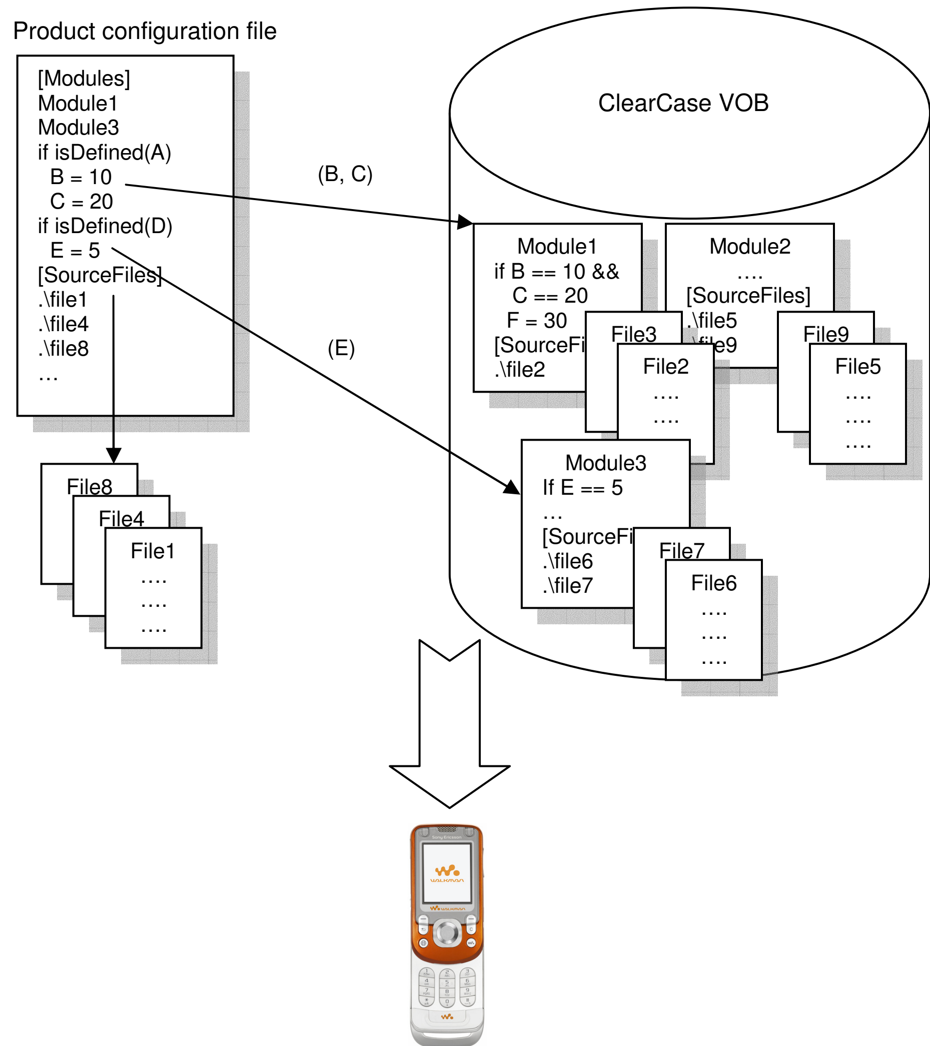


Figure 1. The product, module and configuration variable concept.

Figure 1 displays the relationship between the product configuration file and the modules and their configuration files. The [Modules] section in the product configuration file states which modules the product should use. The configuration variables set in the product configuration file enables and disables functionality, features, in the modules and they also state which source code files that are to be used. The configuration variables set in the product configuration file are passed to the modules, e.g. B, C and E in the figure. The configuration variable F in Module1 is passed to File2 and File3. The configuration is built and the software is loaded into the product, the mobile phone.

Symptoms

With the current way to manage software with interacting features in a modular way and in the same time allow detailed configuration possibilities, these problems, or rather symptoms occur:

- When starting up a new project with new variants, the software does not always build or if it builds it does not always work.

- In order to form a new variant, even if just experimenting, a new product configuration must be set up which requires some work and is time consuming.

- The large amount of configuration variables complicates the overview of the software. The variables are not defined in one place, but spread out in different configuration text files.

- SEMC has defined rules and guidelines on how to configure the software, controlling the creation and usage of configuration variables to enforce the software architecture. Legacy code and configuration variables that do not fulfill the latest software configuration guidelines are not automatically detected and can remain in the software.

Objectives

Decisions of how to address the core problems described has already been made by SEMC and are reflected in the design of the software development environment, but this thesis work aims to further strengthen these efforts.

The thesis work is divided into three parts. Each part has a unique section in this report and the corresponding issues aiming to be solved are further presented in the introductory subsections.

Objective 1: Investigate methods to configure software variants of the phone in a more dynamic way than it is done today and develop a tool to compose and build software variants.

Objective 2: Investigate methods to gather and visualize dependencies between configuration variables.

Objective 3: Investigate how the dependency information can be used, e.g. to form a set of rules to validate configurations.

Method

During the thesis work considerations about the current processes, limitations of e.g. SDE and how the work in general is being conducted at SEMC have been taken into account. To be able to understand how the work is conducted at SEMC interviews have been made, documents studied and presentations have been attended to. These circumstances have meant that the focus has been on internal documentation and routines and that only a few external documents have been useful in the thesis work. A list of recommended reading is presented in chapter 6 in the end of the report.

2 Dynamic Variant Tool

The normal way to set up a new variant is to add instructions to the existing product configuration file, describing how to build that variant. The product configuration file is however under source control and in order to change it, the file must be checked out using the configuration management system.

If there was a more flexible way to create and test variants than today, it would be possible to detect problems that could emerge when setting up new products at an earlier stage. Such a tool would also make it easier to experiment with imaginary variants.

To verify a configuration a build must be performed. Even if the build succeeds it is not guaranteed to produce a working variant. Apart from simplifying the creation of variants, a variant composing tool could also be implemented to perform additional checks than those made at build time to validate the combination of configuration variables.

A tool for setting up variants and building them called Dynamic Variant Tool, DVT, has been created. Dynamic means that the variants are easily set up, managed and built. The tool does not produce configurations meant as final products, they are merely of experimental nature.

Objective

The objective of this part of the master thesis is to investigate methods to configure software variants of the phone in a more dynamic way than it is done today and develop a tool to compose and build software variants.

2.1 Analysis

This subsection will dissect the problem and objective and identify constraints and advantages affecting the choice of solution.

The objective of this part of the thesis work was specified by SEMC on a high level without narrowing the solution possibilities more than necessary. The number of possibilities are however limited because they must be compatible with the current SEMC development environment and standards.

The core task of the tool is to allow SEMC to configure and build a certain variant of their software. Apart from being able to compose a variant, the tool must also be an improvement in ease of use and time consumption over the existing methods. The best current way of composing and building a new variant consists of the following activities:

1. Find out which configuration variables that should be changed and what their new values should be to achieve the desired functionality of the variant. The configuration variables along with their documentation and information on allowed values are defined in configuration files located in different directories accessed through the configuration management system.

2. Edit the configuration files and change the desired configuration variables. This can either be done by defining a new “real” product variant in the product configuration file, or by overriding an existing product using the descrextra file.

3. Build the variant using the SDE functionality. Apart from the produced code a build log which summarizes the build result is generated.

Through an analysis of these activities, the following issues covering ease of use and time consumption are identified

Issues of activity 1: The current practice when setting up a new product is that the person making the configuration is supplied with a list specifying the configuration variables that should be set. Since each variable is created and managed by the developers responsible for a module these are often consulted to supply the correct values. Currently it would be hard for one person to compose a product configuration file on his/her own just by browsing the configuration files spread throughout the source code or its auto-generated documentation. By convention, all public configuration variables must be commented describing their affect and listing the allowed values, but variables with missing comments still exist.

Benefit: A tool can help collect and list available configuration variables, and if available, present the comments of these directly at edit time.

Since the editing of the configuration files is done using a standard text editor, no syntax checks are performed. Syntax errors will not be detected until a build is started or a preprocess is run. Further more, since configuration variables are typeless, the syntax checks performed by SDE do not take into account what type of value the other configuration files are designed to expect from a variable.

Benefit: A tool can perform validation and sanity checks on configurations before a build is started. This can detect additional issues than the ones detected at build time.

Issues of activity 2: The advantage of using the descrextra is that it is not under source control and that it is a view private file. It can easily be edited without affecting the common software under source control.

Benefit: The descrextra files do not need to be checked out to be edited.

Benefit: Descrextra files do not need to hold more information than the distinctions from the original product configuration file to describe a specific variant, thus increasing readability and avoiding redundancy. This however requires good knowledge of the contents of the original configuration files.

Issues of activity 3: It is most likely that there are several variants that need to be built regularly. The tool can provide a graphical user interface that makes it easy to build multiple variants without user interaction.

2.1.1 Tool usage strategy

A variant tool could simplify the following basic activities:

- Setting up variants
- Validating configurations
- Starting builds of specific variants
- Managing and keeping a large amount of variants valid and buildable

With the overall purpose of the tool to reduce the effort and time spent detecting and correcting errors when combining functionality into new variants or starting new product configurations, the following strategy to use the tool is recommended:

Proactively raising delivery quality: The developers responsible for a module compose a set of variants involving their module. This set can consist of all existing products that uses the module plus a number of experimental variants and variants that could possibly become real products in the future. The configuration management department can also supply the developers with suitable variants sensible to interaction with other modules.

Once these variants are composed, they can easily be distributed to the developers responsible for the module and to other developers as well. Developers run DVT whenever they need to see if their code is valid for some or all variants listed in the tool. The tool supports validation and compilation of multiple variants without user interaction and can be run as a batch job during the night.

Before the developers responsible for a module delivers it to the configuration management department they make sure all variants are successfully validated and built by the tool. This will increase the modularity and compatibility of the source code delivered, and the number of problems when building the entire product and later products with different variable combinations are possibly reduced.

Experimenting with new products: Since the tool can provide a flexible way to compose new variants, it can be used when a new product is under investigation. With the tool a user can try variants of existing products to see if they can form the desired functionality and build it.

2.1.2 Possible solutions

The activities and issues identified in the analysis chapter limit the number of possible solutions. In this chapter solutions that fulfill the constraints are presented and investigated. The constraints are:

1. The tool must use the SDE build environment. This is the only build environment available.
2. The tool must follow the current configuration file syntax.
3. The configuration information needed by the tool must be supplied by SDE.

Constraints 2 and 3 imply that source code and configuration files must be accessed from the configuration management system in the same way used during normal development and builds to ensure the received data matches the real source. Practically this means that the tool must gather data from the same ClearCase views it would build the variant from.

Practical limitations affecting the choice of solution to implement in the thesis work are:

- SDE functionality is vital to SEMC software production and access to the SDE source code is limited.
- The time frame of the thesis work is limited to 20 full time weeks. Apart from the implementation of the tool, the thesis work also consists of a theory part, the preparation of this report and an oral presentation.

The risks and consequences of these practical limitations are explained for each possible solution.

Solution 1 - Full SDE integration: The tool is implemented and integrated with the existing SDE functionality. The variant tool needs to access configuration variables and module data from SDE and this solution ensures that changes to SDE does not render the tool useless. Further benefits with this solution are that all data handled by SDE can be made accessible to the variant tool.

This solution is not chosen in the thesis work due to source code access limitation. Apart from this the time to understand the SDE source code and implement the tool would exceed the available time for the thesis work.

Solution 2 - Database: A highly flexible solution can be achieved by creating a database to hold the data needed by the variant tool. SDE gathers all the information needed by DVT and other possible tools and stores it in a database. A benefit of this is that the database would form a well defined interface between the tools and SDE, making them less sensitive to SDE changes. Other database related benefits are that the data is made accessible to other applications than the variant tool, and additional attributes can be added to the database format without affecting the functionality of applications currently using it. This solution requires fewer changes to SDE than the full SDE integration solution. These changes would be to add methods to output the required data into the database format.

This solution is not chosen in the thesis work due to the fact that it requires changes to SDE and that this full scale solution would require more time than available for the thesis work.

Solution 3 - SDE add-on: To avoid changes to SDE source code an add-on solution can be used. Currently SDE can output information about configuration variables and modules in text format using command line calls. This data can be parsed by an external program and used to provide a way of creating variants. Compared to the database solution, this tool uses its own data structures to represent the variables and module data needed. The major drawbacks of this solution is that the parsing mechanism makes it highly sensitive to changes of the SDE output formats and the information accessible from SDE is limited.

This solution is however easier to implement than solutions 1 and 2, and can serve as a pilot project to investigate the potential of a full scale variant tool. If a full scale tool is to be implemented, the source code of the add-on tool can be reused. This solution is chosen for implementation in the thesis work.

2.1.3 Main requirements

To create a tool that fulfils the constraints from the development environment at SEMC and reduces the effort needed to create a variant and build it, requirements where specified. An elicitation process has been conducted and significant effort has been spent to understand the SEMC domain. The main requirements specified can be seen as the result of the previous analysis chapter.

The elicitation process was conducted by brainstorming, interviews, document studies and prototyping. The interviews were conducted with people representing the expected users of the tool. The prototypes were made as paper drawings and represented suggestions of the graphical user interface of the tool. These were reviewed with the stakeholders and redesigned accordingly. From these prototypes requirements were captured. The requirements were then discussed and rewritten with the stakeholders. Both functional and non-functional requirements were specified. A complete list of the functional requirements can be found in [1]. Only the main requirements are presented in this report. Some of the requirements listed are provided with short explanations since the terminology used is sometimes domain specific.

Before the main requirements can be accounted for, the term switchconfig is introduced. The switchconfigs are created using DVT. A switchconfig holds all information about the configuration variables that are changed relative to the original product.

The main functional requirements are:

1. To be able to create a new variant no files shall be needed to be checked out.

2. The tool shall be able to start from the SDE tool menu in Microsoft Visual Studio. This is because the tool needs to access functionality in SDE and the fact that it will be considered to be a part of SDE to the user.

3. Public, private and product configuration variables and their possible values shall be collected by the tool from selected modules.

4. The user shall be able to change the values of public, private and product configuration variables. The collection of edited variables forms a switchconfig which corresponds to a file of the tool.

5. The tool shall be able to generate a descrextra file for a chosen switchconfig. This is because when a variant is built, the descrextra file will be used to communicate the user changes to SDE.

6. A switchconfig shall be able to be synchronized with the ClearCase view. Synchronization means that the tool checks that the configuration variables and their values in the switchconfig still exist and are valid in the view. This is to ensure that the information displayed in the tool is up to date.

7. The tool shall use the build functionality in SDE to verify that switchconfigs are buildable.

8. The tool shall be able to run command line compilations as batch jobs. This is to enable scheduling of variant builds, e.g. as nightly builds.

The main non-functional requirements are:

1. The tool shall reduce the effort needed to create and test a variant. This is actually the main objective of the tool.

2. The tool shall be designed to keep the user input to a minimum. This especially applies to the build part of the tool. The tool shall be able to run as standalone as possible, e.g. during the night.

3. The tool shall be written so that it is easy to maintain and it is easy to add new functionality. This is important since the conditions, e.g. configuration file syntax or the interface of SDE might change.

4. The tool shall guarantee that all modules in the ClearCase view and all the configuration variables in the selected modules are available in the tool. This is to ensure that correct information is given to the user.

5. A user manual shall be written to help the users.

6. Technical documentation describing the tool shall be written. This will be used when future work is done on the tool.

Changes to the requirements: During development the functional requirement number 4, that users shall be able to change all types of configuration variables including private, was changed. The new requirement states that only variables declared public or product shall be able to be edited through DVT. This is due to the fact that private declared variables can not be affected through the product configuration- or descrextra file. At this point there is no possibility to change private declared configuration variables without checking out any files.

The functional requirement number 8, concerning batch jobs run from the command line, was changed. The new requirement only states that compilations shall be able to be run as batch jobs.

2.2 Design

This chapter describes the selected design and some considerations that have been taken into account.

2.2.1 Information access

DVT needs to use functionality in SDE, e.g. for building the variants, gathering information about the modules and configuration variables and starting or stopping ClearCase views. If another implementation had been chosen, e.g. an implementation directly inside SDE this would maybe not be needed. To build the variants the SDE build functionality is used. The information gathered about the modules and configuration variables needs to be processed. At this point DVT is sensitive to changes of the output, e.g. text files, from SDE. There is a risk that DVT will not work if changes are made to SDE that is used as input in DVT.

2.2.2 Data structures

One of the most fundamental parts of DVT is the data structure that stores all information about the modules and configuration variables. Each module can contain configuration variables and each variable can only be declared in one module, but can be used (read) in several modules. For each module the data structure needs to hold the following information; the name of the module and which configuration variables that are declared in that module. For each configuration variable the following information needs to be stored; the name of the variable, its default value, all its possible values and the type of the configuration variable (public, private or product). Since there can be several modules that contains an unknown number of configuration variables and each variable can for instance have several possible values the data structure has to be flexible. The data structure will most likely hold a large number of modules and configuration variables so access to it is required to be fast. The large amount of data also implies that if there are many temporary copies of the data structure it will consume a lot of memory. There are several possible ways to implement the data structure when these requirements and limitations are taken into account. DVT uses a solution with hash maps and linked lists, see Figure 2, which stores pointers to the data. This gives a solution that is flexible, fast and that does not use any unnecessary memory.

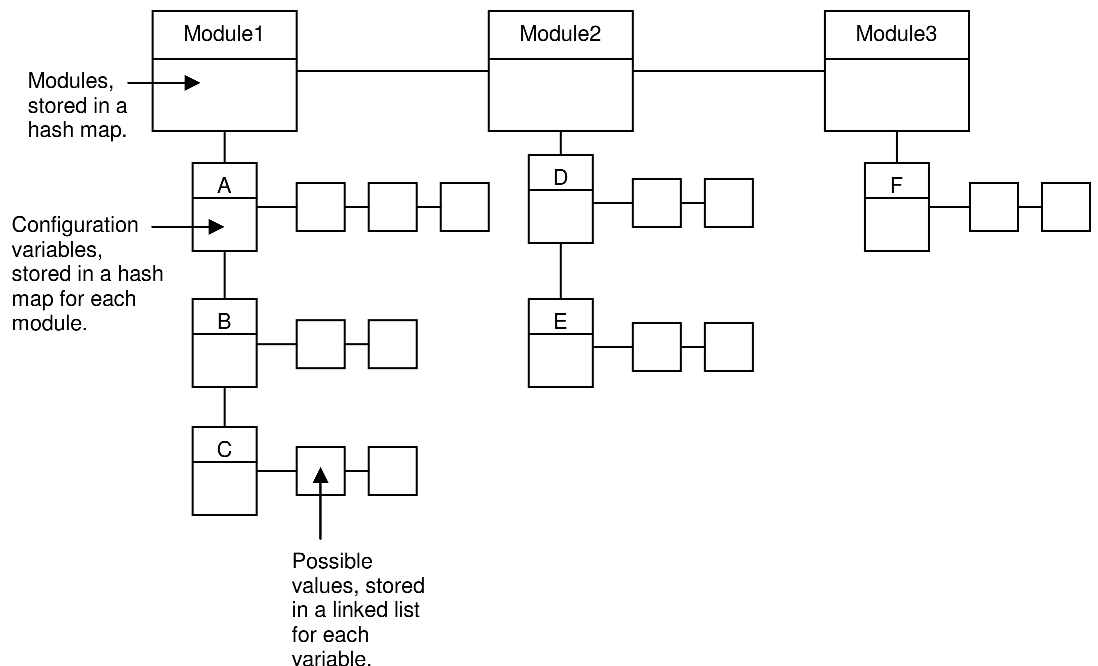


Figure 2. The data structure used in DVT.

2.2.3 Graphical user interface

Since the tool shall be able to build variants as batch jobs and use build functionality in SDE it is a good idea to separate the tool into two parts; one for composing the variants and one for building the variants. The composer holds all functionality for creating and setting up the variants and the builder all the functionality for building the software. The user interface should be functional for all types of users, both novices as well as those that are very experienced. The type of information that the tool needs to handle is suitable to be visualized in list form. The listing also makes it easy to overview the information and operations can easily be executed. The operations can be executed from the menu, toolbar, shortcuts or in some cases by a pop-up menu. The composer consists of two list views, see figure in Appendix C; one for displaying modules and one for displaying the configuration variables, the content of a module, and all the information concerning them. The browser consists of one list view and one build window, see figure in Appendix C. The build window displays the output from the builds. The composer creates a switchconfig file that is used by the builder when the descrextra is generated. The builder supports the building of several variants in consecutive order. The builder creates a compile list file which stores the information about multiple builds.

To keep the user aware of what is going on in DVT the information, processes and communication with SDE run in the background needs to be presented. This can for example be done through the status bar where short messages are displayed. Another situation where it is important to give the user feedback is when something generates an error. The user should not only be made aware of that there has been an error but also what caused the error and even a suggestion of a possible solution.

It is not always certain that the user is familiar with all the configuration variables he or she needs to change. To support the user when working with a variant, information about the configuration variables and their possible values might be useful.

2.3 Implementation

SEMC required that the tool should either be implemented in Perl or C++ due to the fact that it should be easy to maintain since the majority of the tools SEMC has developed are created using Perl or C++. This led to that DVT was implemented using C++ and Microsoft Foundation Classes, MFC. MFC supplies the possibility of simple and quick development of Windows applications. Screenshots of DVT can be found in Appendix C.

2.4 Conclusions

The constraints and limitations presented in chapter 2.1 leads to a method that makes it possible to in a more dynamic way than today configure variants. This method can be implemented using three different approaches; full SDE integration, using a database to store the information about the modules and configuration variables, and as an SDE add-on. These solutions are presented in possible solutions in chapter 2.1.

The selected solution, where DVT is implemented as an add-on to SDE, resulted in a tool which makes it possible to in a more flexible way create and test variants. DVT is suitable to use when new products or variants needs to be set up. DVT consists of two parts; one for composing the variants and one for building them. The build process is used to verify and validate the variants. DVT uses functionality in SDE through commands which outputs text that is parsed by DVT.

Almost all requirements in the main requirements of chapter 2.1 and [1] have been fulfilled. At this point DVT does not support changing variables that have been declared private. The nature of the private declared variables requires that they are handled in a different manner than the public and product declared variables. An example is that there can be several private variables with the same name defined in different scopes. The lack of support for private declared variables in DVT is due to how SDE is implemented and the time constraints of the master thesis work.

DVT has been created to suit both novice and experienced users. However more useful feedback could be presented to the user, e.g. through the status bar or as messages. The data structure holding all information about the modules and configuration variables has been created to be flexible, fast and memory efficient. However, the fact that the data structure uses pointers might increase the risk of memory leakage but efforts have been made to minimize that risk.

At this point DVT is sensitive to changes in the output, e.g. different text files, from SDE. If the SDE output is changed DVT might not be able to interpret it and then DVT might not work as intended. This could be avoided if DVT would be fully integrated with SDE or if there were a defined interface between them.

Future work

Due to the fact of the time limitations of the master thesis work not all functionality was implemented nor could be implemented due to practical/physical limitations. Here are some suggestions of possible future work that can be conducted to further increase the strength of DVT.

To fully be able to change the values of all types of configuration variables support for handling private declared variables must be added. This will make the tool more useful, e.g. when variants of a specific module is to be set up and tested.

To make DVT easier to work with some additional feedback to the user needs to be presented. DVT uses several background operations and to make the user aware of what is going on this could be presented in e.g. the status bar. Another source of improvement is the error messages that are displayed when something causes an error. These can in some cases be extended with further information about the origin of the error and possibly also a suggestion of a solution.

To reinforce the robustness of DVT an interface between SDE and DVT should be declared. No changes to that interface are then allowed to be made without updating DVT to handle those changes. This would make DVT less sensitive to changes in SDE and the risk of corrupt behavior of DVT would be reduced.

DVT was intended to gather and present the comments regarding each configuration variable. The information is gathered from the configuration files. The information explains the purpose and use of the configuration variables. This functionality has not been implemented. This information can make it easier for the user when working with a variant. To further make the configuration process easier for the user it is a good idea if DVT would hold all the possible values for the variables. This information is also gathered from the comments available where the variables are defined. Since all this information, the comments and the possible values, need to be gathered from the configuration files it is recommended that SDE and its parser does this and that the information is then passed to DVT through the common interface.

3 Configuration variable dependencies

Since the software rapidly changes and expands, and the fact that there is a large amount of configuration variables makes it complicated to get a good overview of the software. One objective of the thesis work is to gather and visualize the dependencies of the SDE configuration variables.

Configuration variables and how they depend on each other variables can be represented as directed graphs or flow charts. By visualizing this it is possible to:

- Give an overview of the software. A visual graph is easier to overview than statements in configuration files located in separate directories.
- Analyze the impact of changes in a configuration.
- Validate the configuration files.

This chapter will introduce two structures to hold and visualize information about configuration variable dependencies; the static dependency graph and the dynamic dependency chart. These will have different properties and can be used to analyze the software in different ways.

Objective

The objective is to investigate methods to gather and visualize dependencies between the configuration variables in the configuration files.

3.1 Analysis

The analysis is based on the existing circumstances at SEMC. These circumstances are for example the number of configuration files, the large number of configuration variables and the rapid development of new products that implicitly leads to a large number of variants. The analysis accounts for two different methods of how the dependency information can be gathered and the analysis results in the definition of what a dependency is and a description of the static dependency graph and the dynamic dependency chart.

3.1.1 Definition of dependencies

Affect — Configuration variable A is said to be *affected* when it is either declared, undeclared or assigned a value. The variable is affected at any type of assignment, no matter if the assigned value is constant or composed by other variable values.

Dependency — Configuration variable A *depends* on configuration variable B if either the existence of B or any value of B leads to or is involved in an affect on A.

The consequences of these definitions from a configuration file aspect are that a dependency exists in two cases:

1. Any action on configuration variable A except reading is performed in a scope that is entered by a condition test involving variable B. In this example the configuration variable C depends directly on D and indirectly on B, and A only depends on B.

   a)
   ```
   if B == 10
      A = 20;          //A depends on B
   endif
   ```

   b)
   ```
   if B == 10
      if D == 10
         C = 30;       //C depends on B and D
      endif
      A = 20;          //A depends on B
   endif
   ```

2. A is assigned a value that depends on the existence or the value of B.

   ```
   A = 10 + %B%;        //A depends on the value of B
   A = isDefined(B);   //A depends on the existence of B
   ```

3.1.2 SDE functionality

The SDE preprocess functionality extends the functionality of the normal C compiler's preprocessor and among other things presents a list of defined configuration variables and their assigned values right before the build process is started.

The values of the configuration variables are defined in the configuration files and might differ depending on the different variants. It is during the preprocess that the variables are assigned their values.

3.1.3 Methods for gathering the information

Two different methods have been investigated on how to gather the dependencies from the configuration files; the first one is by an ad hoc approach and the second is by parsing the configuration files.

The ad hoc method works in the following manner; the configuration variables that are of type on/off is first set to hold one value, e.g. on, then a preprocess is run. Next, the variable is changed to the other value, e.g. off, and the preprocess is then run again. The results from the two preprocesses are compared and the difference between them is analyzed and the variables affected by the change are identified. This method is then repeated for every configuration variable of interest.

One limitation of this method is that it only works, at least in a reasonable way, with configuration variables that are of the on/off type, however many variables hold numerical values or strings. It is although possible to use the method even for configuration variables of any type of value, but it is required that all possible values of those variables are known. The fact that a variable can hold many different values causes a combinatorial explosion making it very complex to try all combinations. If only the on/off configuration variables are investigated it is impossible to get a complete overview of all variables and dependencies which limits the use of this method. Another drawback of this method is that it would be time consuming, at least using the current functionality in SDE, to gather dependencies for a large amount of configuration variables and it is recommended that the method is automated since there are many operations that need to be performed.

The second method that has been investigated is the one where the configurations are parsed to find the dependencies. Parsing a configuration file, in plain text, with a defined syntax is a well known operation. Parsing configuration files can be done in two ways; one where the value causing the dependency is taken into account and one where it is not. This results in the dynamic dependency chart and the static dependency graph.

3.1.4 Static dependency graph

If the configuration files are traversed statement by statement from top to bottom, tracking the scopes as if configuration variables had any value possible, every statement will be reached once and all possible dependencies will be collected. These dependencies form the static dependency graph. In a sense this graph presents the worst case dependencies, meaning that a change of a certain variable in the configuration files will at most affect all of its children in the static dependency graph recursively down to leaf nodes. The static dependency graph does not take the values into account. It merely states that there might be a dependency.

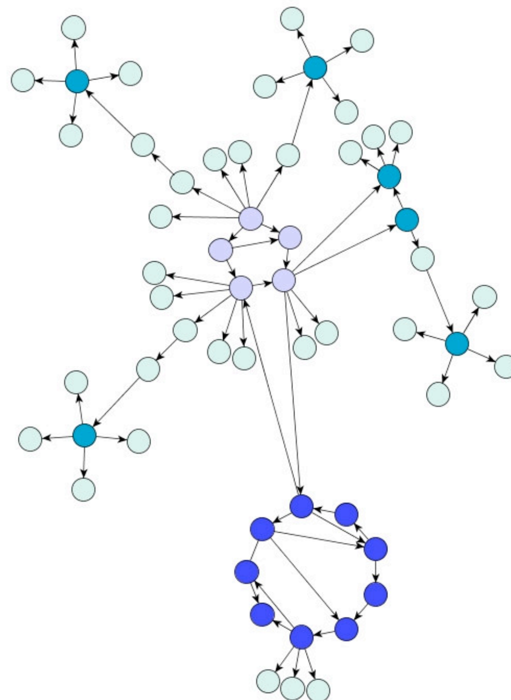


Figure 3. Example of the static dependency graph.

Figure 3 displays an example of a static dependency graph. The nodes represent the configuration variables. Every node is a unique variable and the edges represent the dependencies between them. An edge from one node to another indicates that the first node affects the second one.

The worst case property of the static dependency graph makes it suitable to use if you want to make sure that a change of a configuration variable will never impact on certain variable or modules.

3.1.5 Dynamic dependency chart

To get a more precise representation of the dependencies between the configuration variables their values must be taken into account. The same arithmetic used by the preprocessor must be used to traverse the configuration files when analyzing the dependencies. The purpose of the dynamic dependency chart is to describe all possible paths through the configuration files.

When holding all this information, the dynamic dependencies will form a flow chart rather than a graph. The information stored in this flow chart will be an exact graphical representation of the logical statements involving configuration variables in the configuration files.

The previous configuration code example

```
if B == 10
   if D == 10
      C = 30;
   endif
   A = 20;
endif
```

would result in the following dynamic dependency chart.

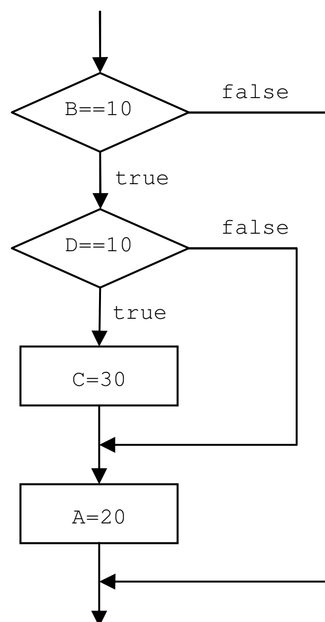


Figure 4. Example of the dynamic dependency chart.

The dynamic dependency chart of a set of configuration files gives a more detailed view of dependencies between the configuration variables than the static dependency graph. The chart in Figure 4 displays under which circumstances A and C are assigned the values 20 and 30.

3.2 Result and discussion

This chapter presents different solutions on how to gather the dependency information. It also presents the outcome of a pilot project performed that aimed to build a static dependency graph from the existing configuration files to allow initial evaluation of if visualization could be useful at all. A suggested solution where the dependency information is stored in a database is presented. Specific issues of making a visualization practically useful are also addressed.

3.2.1 Static dependency graph

The static dependency graph has the advantage of being quite easy to construct from the configuration files. To form the static dependency graph every statement of the configuration files needs to be visited and a top to bottom order of traversal is sufficient. The file order is not important except for the files that are included under condition scopes that involve configuration variables. During the traversal, configuration files will be parsed in search of all types of statements that will lead to dependencies. The following algorithm can be used:

The algorithm uses a scope stack to keep track of which variables that are affecting other variables. The stack holds elements consisting of one or several variables found in if and elseif statements.

1. Get a list of all configuration files used for the product to examine.

2. Gather all available configuration variables from declaration and assignment statements in the configuration files and store them in a list. This list will be used to identify and separate real configuration variables from literals in statements.

3. For each configuration file to examine:

   For each statement in the configuration file:

   Depending on the statement type, do the following:

   if: Add all configuration variables in the statement as a new element and push it to the scope stack.

   elseif: Add all configuration variables in the statement to the topmost element in the scope stack.

   endif: Pop the topmost element in the scope-stack.

   set: If the variable that is set is a configuration variable, it is affected by all configuration variables currently in the scope stack as well as any involved in the value assigned, i.e. in the right hand side of the assignment expression.

include: Recursively perform a dependency check on the included file. The current scope stack is passed as an argument since inclusions can be controlled by condition statements containing configuration variables which means that all variables in the stack at this point also affects the variables found in the included file.

3.2.2 Dynamic dependency chart

The algorithm traverses the configuration files and represents every decision and affect based on configuration variables as nodes in a flow chart. Methods of converting code into flow charts are well known and many programs exist that can convert more complex syntax than the one used in SDE into flow charts.

The task of constructing a parser to obtain the dynamic dependency chart is left outside the limits of this master thesis work. The main reason for this is the fact that a parser external to SDE would be sensible to syntax changes and access to make changes to the SDE source code is limited. It is recommended that the functionality for gathering the dependency information is integrated with SDE. This makes it less sensitive if the syntax of SDE changes and makes it easier to keep up to date.

3.2.3 DVT dependency graph generator

A pilot project using the algorithm for generating the static dependency graph has resulted in an application that constructs a graph of user specified configuration files. This application uses the same data structure as DVT and it has been added to DVT as an extension. The application gathers the dependency information and it is written to a file using a standard graph format. This file is then interpreted by a third party graph editor that handles the visualization. The purpose of this application was to demonstrate how a static dependency graph can be constructed but also to evaluate if and how the graph can be used. From this pilot project several conclusions were made;

- The static dependency graph is easy to create.
- The information it holds gives a very good overview of the software.
- The graph will contain a lot of information if it is constructed from all configuration files. This places requirements of how the information is presented.
- There are still some questions on how the dependency information can be used, e.g. is it necessary to display all the information or is an overview sufficient.

3.2.4 Visualizing the information

If the dependencies are gathered from all configuration files there will be a lot of configuration variables and dependencies to visualize. To be able to work with all this information it is very important how it is visualized. The main issue is not how the information is gathered but how it is presented. There are mainly two ways to visualize the dependencies; the first one is for SEMC to create their own graph editor or they can use one developed by third party. There are several applications on the market that visualizes dependency information. No matter which solution chosen, it is recommended that the editor has to be able to:

- Filter the dependency information. This means that it must be possible for the user to view, filter, a specific module or configuration variable, e.g. display a variable and its children or to display which variables that belong to a specific module.

- Search for a module or a configuration variable.

- Group the configuration variables. For example be able to group the configuration variables using different layouts but also to be able to group them by their properties, e.g. different colors for each module.

- Display the dependency information in different levels of abstraction. This means that it shall be possible to for example display the dependencies of the configuration variables but also the indirect dependencies between different modules that the configuration variables occur in, see Figure *5*. This could also be used with the dynamic dependency chart where it can be used to hide or display certain segments of the chart, see Figure 6 where the information in Figure 4 has been hidden.

- Hold information about where the dependency occurred in the edges; e.g. which module, configuration file and on what line. This especially concerns the static dependency graph where this type of information will be of great help to the user.

The four first recommendations all aim to ease overview of the dependencies and the last one aims to help the user track the dependencies displayed in the graph.

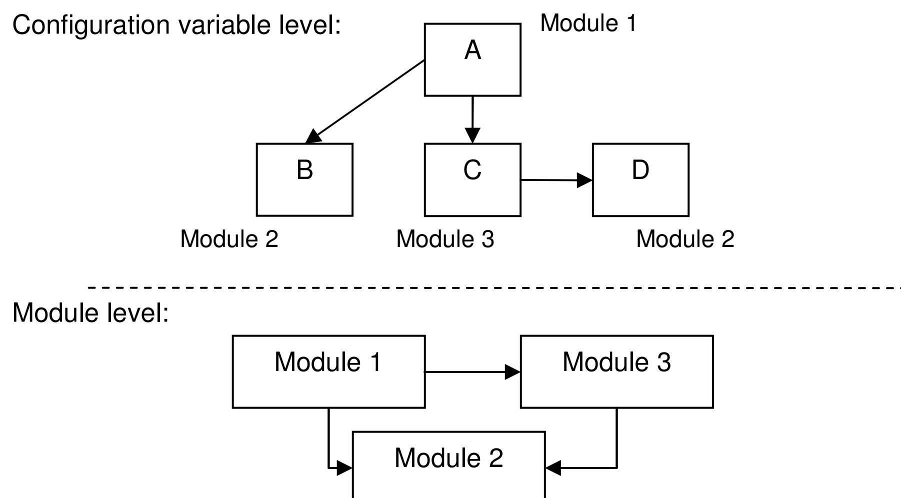


Figure 5. Example of how the dependencies can be displayed in different levels of abstraction.

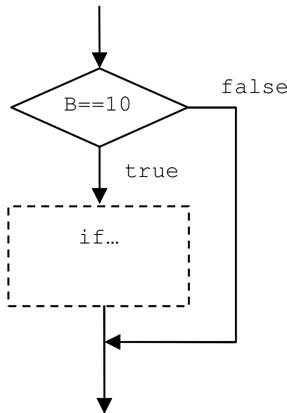


Figure 6. Example of hidden dependency information.

3.2.5 Using preprocess to visualize change impact

If SEMC needs a graphical dependency visualization/analysis program without changing the current SDE functionality the following solution is suggested. Being aware of the limitations of comparing snapshot information, it could still help visualizing changes to users that are interested in the affects of configuration variable value changes, especially if combined with the static dependency graph.

The following algorithm can be used:

1. Gather the static dependency data from the configuration files.
2. Run preprocess for a product configuration.
3. Use the preprocess result to filter all configuration variables not used in the product configuration. This filtering also removes all the variables that have been declared private.
4. Show the final values of the configuration variables from the preprocess in the graph. This gives the starting state of all configuration variables in the graph.
5. The user changes values of the configuration variables that are of interest.
6. Preprocess the new configuration and compare it to the last preprocess results.
7. Present the differences using the graph. A difference can either be an added or removed variable or a value change.
8. Go to step 4. This allows the user to gradually perform and investigate the affects of changes.

The following extensions are recommended to the suggestion:

- The information about the corresponding module, configuration file and line numbers of a node or an edge is stored and displayed to the user upon request. For a node this points to the declare statement of the configuration variable, and an edge would point to all occurrences of affecting assignment statement that cause the dependency. This aims to help the user track the dependencies displayed in the graph.

- Free graph processing libraries exist that can be used to calculate layout and perform drawing. These can be used when implementing a graphical dependency analysis tool. Using this reduces the effort SEMC needs to make in order to create a dependency visualizing tool.

3.2.6 Database solution

If there are several applications, e.g. graph analyzers or a source code diagnostics tool, that need to use the dependency information it might be a good idea to store this in a database. The modules, configuration variables and dependencies can easily be represented using relations. When the dependency information is gathered it can at the same time be added to the database. This would in fact reduce the number of times the dependency information needs to be gathered because every application who wants to access the information would not need to gather it by itself. The biggest advantage however is that a database makes it easy for any application to make queries about the dependency information. This solution also makes it easier when visualizing the dependencies, e.g. when information needs to be filtered or presented in some specific way.

3.3 Conclusions

The objective of this part of the thesis work was to investigate methods to gather and visualize dependencies between the variables in the configuration files. Two different methods have been investigated; the ad hoc method where on/off configuration variables are used, and the parsing method where the configuration files are directly accessed and processed. The ad hoc method has been rejected because of practical limitations. It would almost be impossible to use the method to generate a dependency graph that contains all dependencies. It is recommended that the parsing method is used. When parsing the configuration files two types of dependency graphs can be constructed; the static dependency graph and the dynamic dependency chart. The configuration files consist of a defined syntax and such files are easy to parse.

The static dependency graph gives a good overview of the dependencies and how the software is designed. If the graph holds information regarding where the dependency was found and if it is possible to visualize the graph in a flexible way the static dependency graph will be very helpful and useful in many situations.

The dynamic dependency chart gives the user a very detailed view of the dependencies. It will make it easier to track the exact impact the change of a variable will have.

If SEMC aim to implement the functionality to generate the static and/or dynamic dependency graphs it is recommended that it is fully integrated with SDE. That way it will be easy to maintain and keep up to date.

If the dependency graphs are constructed from all the configuration files they will contain a lot of information. How useable they will be depends on the visualization of the information. It is recommended that effort is made to identify the most appropriate solution that meets the specific needs of SEMC.

Future work

It is recommended that further studies are done, e.g. as pilot projects. During these pilot projects it is suggested that the database solution is implemented, tested and evaluated. It is also recommended that the dynamic dependency chart functionality is implemented, tested and evaluated. An evaluation should focus on whether or not this dependency information can be practically useful at all.

During the thesis work no, or at least very few, considerations of requirements on dependency graphs that SEMC might have has been taken into account. It is now time for SEMC to analyze the results that are presented in this report and then perform further analysis to form their requirements.

4 Framework of rules

The objective of the third part of the master thesis work was specified to investigate how information about dependencies between configuration variables can be used, e.g. to form a set of rules to detect impossible configurations. During the investigation it was discovered that rules can be used to much more than just validating configurations. Because of this, the scope of the third part of the thesis work was shifted to investigate the use and benefits of a general framework of rules.

The situation today is that SEMC has specified guidelines and rules of how to write the configuration files. Typical guidelines cover things like syntax and architectural rules. The guidelines are documented and developers and other people configuring the software are expected to know and follow these. Despite this there are numerous configuration files that does not follow all the guidelines. The main reason for this is that legacy code is used which was written before the guidelines were introduced.

At this point there is no validation mechanism to actually enforce the guidelines. Such a mechanism is not intended to replace the guidelines documentation, but actively helping people to follow the guidelines can be a very powerful method to raise the quality and keep the large amount of code and configurations maintainable, which is absolutely critical in a large and global organization like SEMC.

This part of the report presents the analysis made on possible benefits of a framework of rules. The exact design of the framework is largely dependant on the desired functionality and requirements from SEMC and to form these, thorough investigations must be made. This part of the report can be seen as the first step of such an investigation.

Objective

Investigate if and how a framework of rules can be used to verify and validate configurations.

4.1 Analysis

The framework of rules should be able to facilitate a set of rules used to check the configuration files. The framework must be designed to be flexible and easily support maintenance and adding of new rules. A rule can e.g. be a specification of how to detect illegal access to private configuration variables or simply how to detect illegal value assignments.

Anyone adding, removing or changing the value of a variable or in any other way changing the configuration files could use the framework to validate the changes made. Hence, people all the way from development to the configuration management department could directly use the framework and its validation mechanism.

4.1.1 Rules

There are several ways to collect and form rules. Some are already defined through SEMC guidelines and definitions and other rules can be composed as they are needed. The rules that should be specified can be of varying complexity. Some may require powerful algorithms that must be implemented using a programming language or even run SDE external programs, while less complex rules can be specified using simple script languages.

When adding a rule it is important to strive to make it as maintenance free as possible so that the effort to keep the framework up to date will be as small as possible. It is for instance a good idea to define patterns and syntax of how to specify rules. An example of this can be illustrated with the case of a rule that checks that configuration variable naming conventions are followed. A pattern for naming a variable can e.g. be <IDENTIFIER>_<TYPE>_<MODULE>_<FEATURE>. With a specified pattern like this, the actual algorithm to perform the check is easily implemented.

However the exact details of how the rules should be implemented are left for SEMC to investigate and decide.

4.2 Result and discussion

The investigation has resulted in advice on how and where the framework can be used and issues that need to be considered when designing and implementing it. A recommendation on how rules can be categorized and grouped in the framework is presented.

4.2.1 Strategy

This subsection presents suggested strategies for specific situations where the framework might be useful.

Enforcing configuration guidelines: Making people aware of that they are violating a guideline might affect them to start following these. When new guidelines are being introduced, rules to enforce them can be added to the framework. Warnings and errors caused by the new rules will effectively announce the changes and make people read and understand the guidelines documentation.

Supporting good practice: If the situation occurs where developers introduce bad practice to the configuration files it can be a good reason to add a new rule to the framework that detects the bad practice.

Pre-build validations: The build time of the software is often relatively long, which delays the development work. SDE could be set to automatically run selected validations using the framework as an initial step in builds. The framework can even hold rules that detect errors normally not detected until after a while in the build. A limit can be set which cancels the build command if to many or to critical issues are detected in the pre-build validation performed by the framework.

Metrics collection: The result from validations can be summarized to form quality metrics for projects. The number of errors and warnings acceptable can be defined as non-functional requirements for a specific product. It can even be made possible to specify which rules that are of special importance for the quality of a certain product. When rules are specified, they can be assigned a quality weight attribute, which can e.g. be multiplied with the number of errors detected by that rule to calculate the total quality reduction.

Extending the code diagnostics: The code diagnostics tool that is used to analyze the source code can be extended to run and present selected rule checks from the framework.

4.2.2 Implementing the framework

To be able to perform checks on configuration files, additional input might be needed. An example of a rule that requires external input is an architectural rule to verify that a module does not access architectural layers restricted to the module. The input needed would be the layer structure definitions. The type of input varies depending on the rule and some rules might use the same input. The input can be specified in the configuration files or supplied through other methods, e.g. by xml files.

To be able to perform a check on a configuration file a checking functionality has to be integrated with SDE or constructed as a separate application. There are several different solutions for implementing the framework in a separate application. The application can be written in a programming language like C++, some script language or a SDE syntax parser can be constructed. However the most preferable solution is that the framework is integrated with SDE. The main reason is the same as mentioned earlier in the report for DVT and the visualization tool; keeping the code integrated with SDE avoids double maintenance and the framework is sure to be updated when changes are made to SDE.

Another issue that needs to be further considered is the distribution of the framework and access rights to it. Who should be able to change or add rules? Vital rules could be made protected while others can be editable by anyone. This is a company policy question rather than an implementation issue.

The people changing the configuration files must have a fair chance to be able to keep the configuration files valid according to the rules. To avoid flooding the users with errors and warnings, various techniques can be used. One is to set a limit of the number of errors and warnings to report for each rule and group of rules. Another technique is to be able to set the priority (weight) of a rule. This makes it possible to redefine the importance of a rule as the number of violations changes. For example when the number of errors on a certain rule decreases to an acceptable level, the importance of another rule can be raised. This mechanism makes it possible to gradually tune the quality of the software. There are several ways to define the priority levels of rules. The number of levels can be quite course (e.g. high, as usual and low) or they can be fine grained (e.g. a 1-100 scale).

4.2.3 Rule categories

To make the framework easier to work with it is recommended that the rules are categorized. Reasons to categorize are to improve overview and simplify the implementation. The categorization can be based on for instance how rules are implemented, what property it validates or the input used. Since the framework should support possibilities to run selected checks, the categorization could also serve as a convenient way of disabling or enabling groups of rules. This subsection presents a suggestion of categorization.

At the moment six categories are suggested: functional, physical, syntactical, architectural, recommendations and optimization.

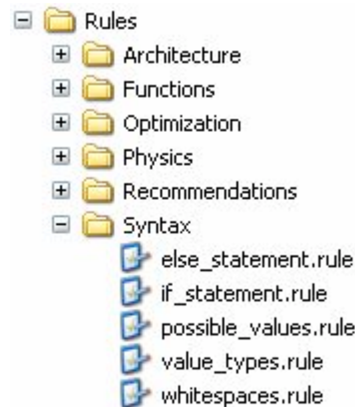


Figure 7. Framework of rules example.

Figure 7 shows an example of a framework of rules. The folders represent categories and the rule files represent specific rules.

Functional

Some of the functionality and features in the mobile phone can be switched on or off through the configuration files while some features are always used. To be able to disable or enable a feature when configuring, a feature configuration variable must be defined. By keeping track of the feature variables and how other variables depend on or affect them it is possible to determine the dependencies between features. This makes it possible to detect impossible configurations. It is recommended that the list of feature configuration variables and how real features depend on each other are specified in a separate document, e.g. an xml file. The xml file would act as input when the check is performed. A typical example of a dependency between two features is that the MMS feature requires WAP to work.

Physical

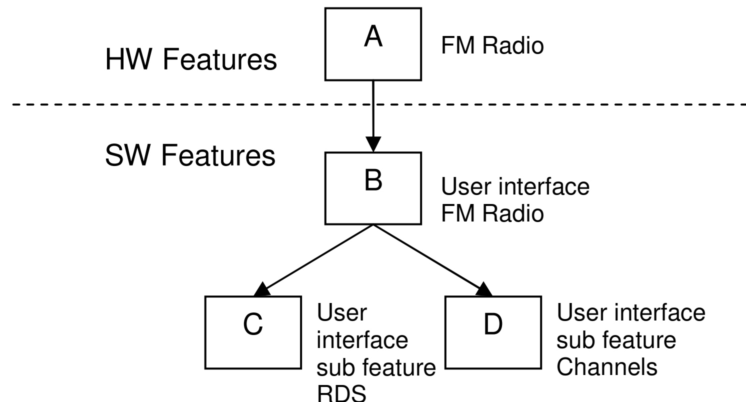


Figure 8. An example of the distinction of features and hardware features.

SEMC makes a clear distinction of software and hardware features. When a feature depends on the presence of some hardware there is a physical dependency, e.g. it is impossible to switch FM radio on if there is no radio receiver hardware present. To clarify the difference the user interface functionality for the FM radio would be classified as a regular feature while the FM Radio code is considered a hardware feature as shown in Figure 8. It is recommended that the specification of hardware feature configuration variables and the real hardware dependencies are listed in the same file as the features in the functional category.

Syntax

These rules are used to validate the syntax of the configuration files, e.g. how a variable should be declared and if the value of a variable is valid. If the configuration variable can have the values 128 and 256 it should be impossible to set it to anything else. The input in this case is gathered from the code comments for each declared variable. Possible values can also be automatically gathered in the configuration files. This is done by checking all if-statements involving the variable and what values are expected from it. Not all if-statements compare configuration variables with static values, so the automatic check can never guarantee that all possible values are identified. Still any possible value found is useful if the configuration variable is uncommented.

The syntax rules and checks can be used to more than just validating statements. It can be extended to verify that configuration variables are commented according to the guidelines, thus improving the documentation quality.

Architectural

These rules relate to architectural issues. Most of these rules would be defined by the software architecture team. The framework can e.g. check for violation of the layer structure. The typical input for such rules is the layer permissions and structure files.

Recommendations

Rules in the recommendations category are not intended to find errors. The purpose is to enforce good practice. The recommendations aim to ensure that the understandability of the configuration files is high and that things are done in a unified way. An example where understandability problems are to be detected is to check the depth of if-statements. If the depth is too deep it is likely that the configuration file is difficult to understand. The external input to the rule in this case is the maximum dept allowed.

Optimization

The optimization rules aim to find possible improvements in the configuration files. Improvements can be made by e.g. detecting unnecessary included source code files to reduce link times and reduce the size of the generated code.

4.2.4 Possible drawbacks

A framework of rules can be a powerful mechanism to raise the quality of the software, but there are some possible drawbacks and risks that needs to be considered and further investigated.

False security: When automating validations there is always a risk of creating false security. Just because a configuration passes all rule checks, it is never guaranteed that the configuration is valid. This risk is reduced by keeping the users aware of the limitations of the rule checks. A practical tip is that the way the validation results are presented can be designed to point out that no known issues were detected rather than stating that the configuration is valid.

Limiting design freedom: The rules must not limit the choice of design more than necessary. Developers can feel that their creativity is held back and new ways of improving things can be missed. The mechanism presented to allow weighted priorities and limiting of the rule sensitivity is a good way to dynamically avoid this problem.

To hard to maintain: The framework must be designed carefully to minimize maintenance and allow flexibility. If SDE functionality changes often and the framework does not handle these changes the cost of maintaining it can exceed its benefits.

4.2.5 Economical analysis

Analyzing the cost and benefit of the framework of rules over its life cycle, some interesting issues are detected. In the analysis it is assumed that issues detected by the framework are solved so that the rule was profitable. If the benefit can be expressed using the same unit as the cost, the following principle chart can describe the economical life cycle of the framework of rules.

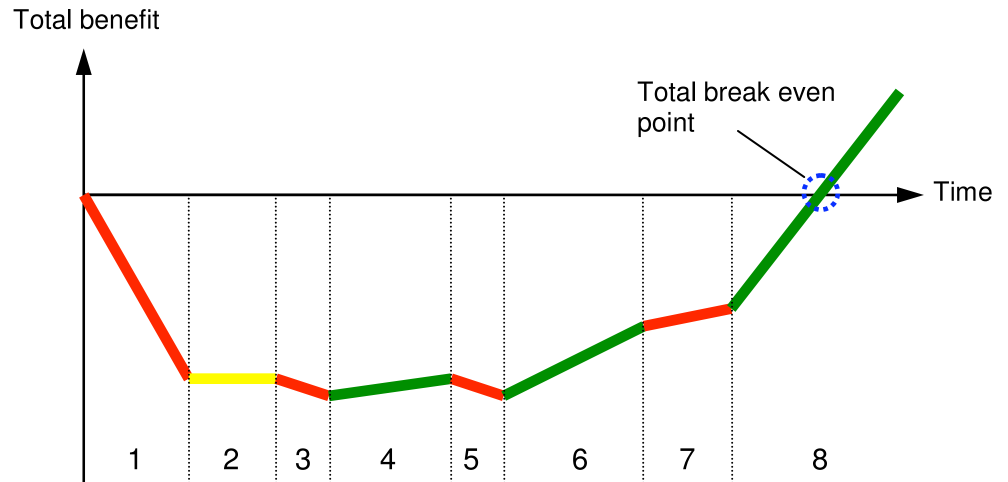


Figure 9. Economical life cycle of the framework of rules.

The different stages of Figure 9 are:

1. The framework is implemented. The manpower needed and time spent affects the slope and duration of this period.

2. The framework is done but no rules exist. This is to illustrate that the cost and benefit of having the framework is zero when not using or updating it.

3. The first rule is implemented.

4. Immediately after the first rule is ready and checkable the framework can start returning benefit. This is where the flexible design pays of when compared to software that needs to be completely done before it can start returning benefit.

5. From this point on, every rule that is added affects the total benefit and cost of the framework of rules project according to Figure 10. Every added rule adds benefit, and speeds up the total return rate of the project. This goes for point 6 to 8 in the figure as well.

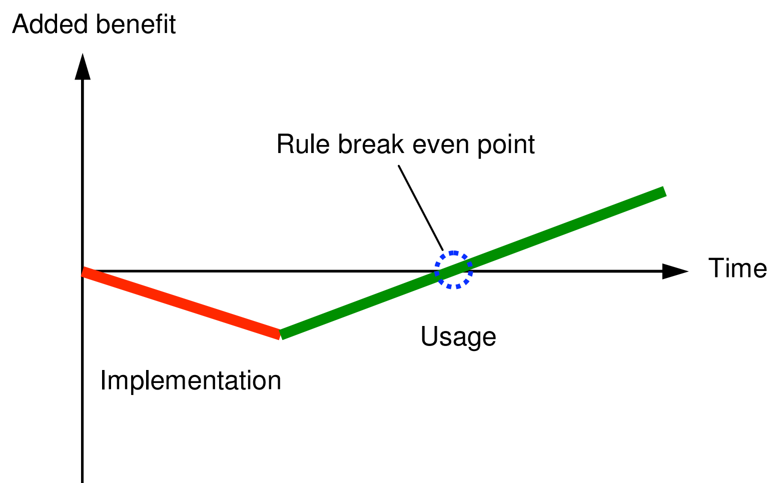


Figure 10. Cost and benefit of adding a rule.

Figure 9 shows a simplified and successful scenario because the break even point is reached. This may not always be the case. Three major factors affect the economical outcome of the framework of rules project: Maintainability and correctness of the implementation, and the benefit of each individual rule. As for maintainability and correctness, at any point there is a risk that errors are found in the framework and in its rules. Every maintenance will push the break even point forward in the time scale. Another effect of a discovered defect is that the predicted benefit return rate has actually been incorrect up to the point where the defect was detected and will be accurate only after the problem is resolved.

The key strategy when prioritizing and planning what rules to implement and in what order is to identify the rules that raise the quality of the software the most compared to the implementation costs and predicted maintenance of it. This is not the same property as the rule break-even point. The most powerful rule can in the same time be to difficult to implement thus making it useless. Furthermore, the benefit and implementation cost of rules are not individual, for example rule A+B separately is not always equal to A+B combined.

As with all quality improvement measures there is a delay on the return of the benefits. The benefits introduced by the framework are actually not received until the affected product has reached the customers and the improved quality has affected the sales figures. This makes the real economical life cycle chart for the framework of rules even more complex.

4.3 Conclusions

By introducing and implementing a framework of rules SEMC has the opportunity to further raise the quality of the configurations files, source code and in the end the quality of software in the mobile phone. The investigation presented in this report is the first step of the evaluation process that needs to be made before a framework of rules can be designed and implemented. The potential of a framework of rules is not limited to just configuration files and can for instance be extended to perform rule checks on source code as well.

To make the framework practically useful, its credibility must be kept high. This is achieved by controlling the degree of sensitivity to a reasonable level depending on the current quality of the configuration files. Flexibility and maintainability is absolutely vital to make the framework useful.

Concerning the implementation of the framework, it is recommended that the framework is integrated with SDE so that it will be accessible to any SDE user. The implementation should be close to SDE source code to avoid double maintenance, but the formats and access rights of the rules are left to be investigated.

Future work

Continue the investigation and further address the issues presented in this chapter. It is recommended that the investigation starts by focusing on whether the framework could be useful at all, and if the best case occurs where the framework is successfully implemented and used, how long can it exist? Will the entire configuration mechanism at SEMC change and render the framework of rules useless?

5 General conclusions

The main objectives of the master thesis work are; investigate methods of how configurations can be made in a more dynamic way, how dependencies between configuration variables can be visualized and how a framework of rules can be constructed and used.

The result of the first objective, to create a tool that makes it possible to create variants in a more dynamic way, is a tool called DVT. DVT makes it possible to create variants without having to create a new product. This is achieved using a descrextra file that contains all the affected configuration variables and their new values. DVT does not require any files to be checked out from the configuration management system. DVT displays the configuration variables for each module and it is easy to change the value of a variable. Variants created by DVT are derived from views and variants from existing products. Before the build process is started the descrextra file is generated and it is then used in the build process. DVT makes it easy to build and rebuild the created variants, it is also possible to do this as batch jobs that e.g. can be run during the night.

During the analysis of the second objective, how to visualize dependencies, it became apparent that there are different types of dependency information that can be visualized. Two types of dependency graphs where defined; the static dependency graph and the dynamic dependency chart. The static dependency graph shows all dependencies between the configuration variables in the configuration files and do not take the values of the variables into account when constructing the graph. It just shows the mere existence of the dependency, a worst case view of the real dependencies. By viewing this graph it is possible to say that a configuration variable A in some case affects a variable B, but it is not possible to say in which situation and how. The dynamic dependency chart displays all possible values of a configuration variable and the path through other configuration variables that needs to be followed to result in a specific value. This gives a very precise view of the dependencies. There are several possible options on how to implement an application that constructs the graphs. We recommend that the graph generating functionality is integrated very closely with SDE to avoid that any unnecessary work has to be performed if the syntax of the configuration files changes. Another reason is that SDE already holds the information needed to create the graphs, it just has to be processed and presented.

The framework of rules shall consist of a set of rules that are used to validate and verify the configuration files. We suggest that the rules are categorized. Six different categories are presented; functional, physical, syntactical, architectural, recommendations and optimization. If the rules are divided into categories it will be easy to check only a selected set of rules. There are several other reasons to categorize the rules, e.g. that it makes the implementation of the rules easier since some of them might use the same input data or just that they are similar so they can be implemented in similar ways. The report also suggests that it shall be possible to separately check the configuration files and not only before a build is started. It is recommended that the framework of rules is integrated as closely as possible with SDE. This will make it easy to use, maintain and keep up to date. The framework of rules can be used in several situations and circumstances to raise the quality of the configuration files.

This report has addressed several issues of improving the software configuration environment at SEMC to help the company produce modular and highly configurable software while on the same time supporting production of software with advanced behavior.

We are certain that major benefits can be achieved but the risks and costs of reaching these are still to be carefully and thoroughly investigated.

6 References and recommended literature

1. A Hellström & B Pileryd (2005). Requirements list – Dynamic software configuration: 2/006 51-LXE 108 010 Uen, internal SEMC document.

2. P Miller (1998). Recursive Make Considered Harmful, AUUGN Journal of AUUG Inc., pp. 14-25.

3. H Zhang & S Jarzabek. XVCL: A mechanism for handling variants in software product lines, School of Computer Science and Information Technology, RMIT University, Melbourne.

4. T Kojo, T Männistö & T Soininen. Towards Intelligent Support for Managing Evolution of Configurable Software Product Families, Software Business and Engineering Institute of Technology, Helsinki University, Helsinki.

Appendix A – Abbreviations

DVT	Dynamic Variant Tool
MFC	Microsoft Foundation Classes
SEMC	Sony Ericsson Mobile Communications AB
SDE	Software Development Environment

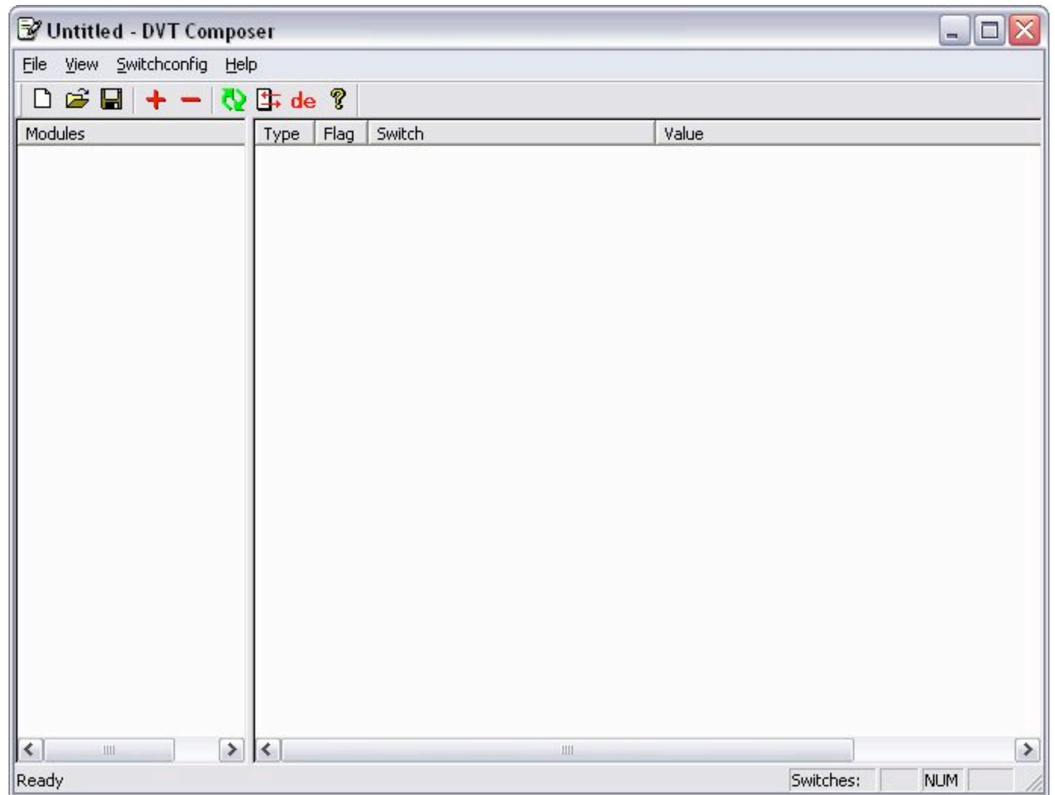
Appendix B – Terminology

Compile list	A list of switchconfigs that specifies the variants that are to be built using the DVT builder.
Configuration file	A file that contains information about whether or not software functionality shall be enabled or disabled. The configuration file also specifies which modules and software source code that is to be used. The configuration file is used in SDE. Each software module has its own configuration file and there is also one configuration file for the entire product.
Configuration variable	A configuration variable is a mechanism in the configuration file to enable or disable functionality in the software. A variable has the same properties as a normal programming variable in terms of available actions, which means that a configuration variable can be declared, undeclared, read or assigned a value. Configuration variables are type-less and no warnings will be issued if a variable expected to hold a numerical value in the configuration are assigned a character. There are three types of configuration variables; public, private and product. A variable can be declared booth public and product.
Descrextra file	SDE uses an extra configuration file, which is only used in the private build of a software product. This file is called the descrextra. The file can easily be modified to list files and/or modules that are to be added to or removed from the product. It can also be used to create, remove or change the value of configuration variables.
Module	A module groups functionality, source code. Each module has its own configuration file.
Product	A product is in the end the mobile phone itself and it can exist as different variants of the hardware as well as the software.
Preprocess	The SDE preprocess functionality extends the functionality of the normal C compiler's preprocessor and among other things presents a list of the defined configuration variables and their assigned values.
SDE	SDE is the main development and build environment at SEMC. It is a system that implements a generic software development environment. It provides a unified interface to software development and configuration management tools. SDE also manages the complete build process for arbitrary software products and software modules.
Snapshot	The values of the configuration variables given a specific point in time is said to form a snapshot.
Switchconfig	The switchconfigs are created using DVT. They hold all information about the configuration variables that have

	been changed relative to the original product.
Variant	A variation of the software developed to provide slightly different functionality than the original version, e.g. adapted to different markets or operators.

Appendix C – Screenshots of DVT

Screenshot of DVT Composer



Screenshot of DVT Builder