# Package Dependency Structures

## - On the use of Debian and APT for software package management in Sony Ericsson

LUNDS UNIVERSITET
Lunds Tekniska Högskola

**LTH School of Engineering at Campus Helsingborg**

Bachelor thesis:
Sven Selberg

LTH School of Engineering
Lund University
Box 882
SE-251 08 Helsingborg
Sweden

LTH Ingenjörshögskolan vid Campus Helsingborg
Lunds universitet
Box 882
251 08 Helsingborg

Printed in Sweden
Media-Tryck
Biblioteksdirektionen
Lunds universitet
Lund 2012

## Abstract

Dividing complex systems into smaller components that are easier for humans to comprehend and manage, is a wide spread method in software design. As systems grow more complex, the task of solving the different dependencies and constrictions of the various components in a system, and create a stable and valid composition naturally also gains complexity. Various tools have been constructed to manage these components, one of which is APT-get, the meta-installer used by Debian to resolve dependencies and constrictions on the Debian package.

APT-get and the Debian package are used in Sony Ericsson's composition system. The dependency-structure is starting to reach the level of complexity where APT-get's limitations are beginning to reveal themselves. A revision of the dependency structure and the tools used for composition therefore was a necessity, to lay the ground work for changes in the composition system.

This bachelor thesis handles three aspects of such a revision:

- A requirements specification of the requirements on dependency structure in the future.
- A tool for easy revision of the present dependency structure.
- An investigation into possible alternative meta-installers and/or alternative component-management-systems.

The result is a requirement specification of relevant stakeholders requirements on package dependencies, with an implementation suggestion using the present package management system. An investigation of alternative meta-installers led to that Sony Ericsson is now pursuing a course set by research conducted by the consortium Mancoosi, and are creating their own resolver based on reverting the problem into a Boolean problem that can be solved by a SAT solver. A fully integrated tool for dependency structure visualization and dependency graph traversing was implemented using Java.

Keywords: Debian, package, dependencies, Sony Ericsson, APT-get, graph, requirement specification, meta installer, solver

## Sammanfattning

Uppdelning av komplexa system i mindre komponenter som är enklare för människor att förstå och hantera är en vanlig metod i mjukvarudesign. När komplexiteten i ett system ökar blir uppgiften att administrera beroenden och konflikter för att säkra en giltig och stabil komposition i samma system även den mer komplex. En mängd verktyg har konstruerats för att hantera dylika situationer. En av dessa är APT-get, metainstalleraren som används av Debian för att hantera Debianpaket.

APT-get och Debian används av SEMC (Sony Ericsson Mobile Communication) för mjukvarukomposition. Beroende- strukturen i detta system har nått en sådan grad av komplexitet att APT-get's begränsningar har börjat visa sig. En revision av beroendestrukturen och de verktyg som används vid mjukvarukomposition, för att användas som underlag för framtida förändringar, är därför en nödvändighet.

Detta examensarbete hanterar tre av aspekterna av en sådan revision:

- En kravspecifikation som specificerar krav på framtida beroendestrukturer.
- Ett verktyg för att undersöka nuvarande och framtida beroendestrukturer.
- En undersökning om möjliga, alternativa, metainstallerare och/eller alternativa pakethanteringssystem.

Resultatet är en kravspecifikation med krav på en framtida beroendestruktur från relevanta intressenter, med ett förslag till hur kraven kan implementeras med nuvarande pakethanteringssystem. En undersökning av alternativa metainstallerare har lett till att SEMC valt att fortsätta med APT-get med en extern beroendelösningsalgoritm, baserad på forskningsresultat från Mancoosi, som omvandlar problemet till ett boolskt uttryck och löser det med en SAT-lösare. Vidare har ett fullt integrerat verktyg för visualisering av beroendestrukturer och traversering av beroendegrafer implementerats med hjälp av Java.

Nyckelord: Debian, Debianpaket, pakethantering, beroenden, Sony Ericsson, APT-get, grafer, kravspecifikation, metainstallerare

## Foreword

This bachelor thesis stems from Sony Ericsson Mobile Communications Lund’s wishes to analyze the present, and possible future, dependency structures in their software composition. And investigate alternatives to the present package management tools.

The work has been conducted on site at Sony Ericsson Mobile Communications (SEMC) in Lund.

The work has been done in cooperation with the staff at SEMC Mobile Communication Software Environment department, in particular with the assigned mentor, Axel Bengtsson, leader of the Build and Composition team at SEMC Lund.

I would like to thank: Axel Bengtsson for being a great support and bottomless source of inspiration. Clas Thurban, Project Manager at Software Environment for valuable input and support, Andreas Larsson Manager at Software Environment for giving me the opportunity to explore the composition system at SEMC, my supervisor at LTH Mats Lilja, my examiner Christian Nyberg and last but not least fellow student Kristian Månsson for making it seem possible...
Sven Selberg

## List of contents

# 1 Introduction

A plethora of solutions and tools exist for automating the retrieval, configuration and installation of software packages on Unix-like computer systems. One example is APT (Advanced Packaging Tool) which is based on the Debian software package format, used in well-known distributions such as Debian and Ubuntu. Based on the same purpose and principles, SEMC uses APT and Debian packages for composing the SW used in their products.

## 1.1 Objective

A complex multi-dimensional package dependency structure is pushing this technology to an extent where its limitations are starting to show, in turn introducing a high risk in the software production chain. In order to manage this risk, a good understanding is needed for the currently used and potentially upcoming package dependency structures. And, for these structures, SEMC would like to know how well Debian and APT perform in comparison to alternative solutions.

## 1.2 Scope

The scope of this master thesis is thus to:

- Understand the Debian software package format and associated tools.
- Create a tool for graphical visualization of Debian package dependency structures.
- Identify SEMC requirements on package dependency structures.
- Identify strengths and weaknesses in the SEMC solution compared to possible alternatives.

## 1.3 Purpose

The purpose of this bachelor thesis is to:

- Visualize the present dependency structure.
- Specify the requirements on the future dependency structure.

And to;

- Find alternatives to the present dependency management system.

## 1.4 Problem description

### 1.4.1 Tool implementation

Making a tool that:

- Can be integrated into SEMC’s present tools and systems.
- Uses data from metadata or build-logs to visualize graphs.
- Visualizes dependency structures.
- Makes it possible to traverse dependency graphs.

- Investigate the metadata of selected packages.

#### 1.4.2 Requirements

- Identify vital stakeholders, within SEMC's organization, and their requirements on dependency structures.
- Visualize these requirements in a specification to aid their use in future dependency structure remodeling.

#### 1.4.3 Finding alternatives to present package management

The objective is to dentify the strength and weaknesses of the present package management and finding valid alternatives.

### 1.5 Methodology

Given the nature of the tasks of this thesis, it did not lend itself to applying a given method. The method was instead to:

- Acquire a solid knowledge-base through extensive research on the subject of Debian packages, dependency structures and dependency solving.
- Apply acquired knowledge whilst implementing the different features.

The different methods used to implement the different parts of this thesis is described in the chapters concerning those features.

### 1.6 Limitations

#### 1.6.1 Implementation

The overview visualization is limited to the present dependency structure due to the NP completeness of the graph layout problem (Joseph Diaz, 2000).
The implementation concerns the: depends, conflict and provides relationship, with support to expansion to include all of the relationships in the Debian metadata specification (Jackson & Schwartz, 2011).

#### 1.6.2 Requirements

The requirements are restricted to those directly concerning package dependencies.

#### 1.6.3 Finding alternatives to present package management

Since there is close to infinite different package management setups the search is limited to an effective time of two work weeks (80 hours) and is to be deemed concluded by the end of this scope or when a valid alternative is found and accepted by affected parties at SEMC.

# 2 Technical Background

This chapter gives a short background to the work presented in this thesis.

## 2.1 GNU/Linux

In 1984 the GNU project started with the ambition to create a free UNIX like operating system (Stallman, 2012). GNU stands for “GNU’s Not Unix”. By the early 90’s all that was missing for GNU to be a functioning operating system was the kernel. The GNU project’s own efforts to create a kernel, the GNU Hurd, this proved to be a time consuming task and when Linus Thorvald freed Linux in 1992 it was the last piece of the GNU system. It was now possible for users to combine Linux with the GNU system to get a free operating system, a “Linux based version of the GNU system; the GNU/Linux system, for short.”.

In the beginning of GNU/Linux the user had to compile each program that they wanted from the source code given to them in a *.tar.gz file.

## 2.2 Debian

The Debian Project was started by Ian Murdoch in 1993. The idea was to create a free distribution of  the GNU/Linux system. A distribution was “micro packaged” which is to say that the software were packaged in several packages with detailed meta information about inter-package relationships. The famous Debian package was born. Users no longer had to unpack and compile source code from various files (Garbee, 2011).

## 2.3 DPKG

With the creation of Debian a tool for handling the unpacking, installation and removal of the packages on the system was created, this tool was dpkg. It is responsible for unpacking single packages (Debian Team, 2011).

## 2.4 APT-get

Apt, or Advanced Packaging Tool, was created to deal with the complexities of package management (Debian Team, 2011). It is a meta-installer, which is to say it tries to solve all dependencies of an installation, after which it relays these packages to dpkg who installs the individual packages. Apt-get is not restricted to handling installation of software, it also handles upgrades and removals in a way that keeps the system stable. Apt-get is explained more thoroughly in chapter 5.

# 3 Debian Binary Packages

At the core of this thesis lies the Debian package, here's a short specification of what a Debian package contains, and an in depth explanation of the binary package control file (Jackson & Schwartz, 2011).

## 3.1 Debian Binary Package format

A debian package consists of an (.ar) archive with 3 files (debian-binary, control.tar.gz, and data.tar.gz):

- Files to be installed on the system when the package is installed (data.tar.gz).
- Control information files(control.tar.gz)
    - The binary package control file containing the control fields of the package (control).
    - Package maintainer scripts (preinst, postinst, prerm, postrm...).
    - A file for shared library dependency information (shlibs).
    - A file that lists the package configuration files (conffiles).
    - A file containing the MD5 sums for the files in data.tar.gz.

The control-file within control.tar.gz is mandatory, the rest are optional.

## 3.2 Binary Package Control File (Package metadata)

The control file contains vital information about the binary package and consists of these fields:

- Package – The name of the binary package (mandatory)
- Source – the name of the package. May be followed by the version number in parentheses. The entire field may be omitted if the source package has the same name and version number as the binary package.
- Version – The version of the package (mandatory).
- Section – the application area to which the package has been classified, i.e. "admin", "database", "games", etc. (recommended)
- Priority – the priority value of the package, i.e. "required", "important", "standard" etc. (recommended)
- Architecture – the architecture which the package is meant to run on (mandatory)
- Essential – a Boolean value field, if set to "yes" the package management system will refuse to remove the package.
- Relationship fields (described below)
- Installed-Size – an estimate of the total amount of disc space required to install the package
- Maintainer – name and email address of the person or group of people that are responsible for maintaining the package (mandatory)

- Description – text describing the package, consists of  two parts, a synopsis and a longer description (mandatory)
- Homepage – the URL of the website for this package

## 3.3 Relationship fields

The information provided by these fields is at the core of this bachelor thesis. For an in-depth explanation of the relationships these fields represent see below.

The relationship fields described below can be divided into three different categories:

- Those who list the packages that are needed for the core functionality or the packages that enhances the functionality of the given package (depends, recommends, suggests, enhances, pre-depends).
- Those who list the packages that cannot be installed together with the given package (conflicts, breaks).
- *Provides*, which provides a virtual package, normally describing one or several functionalities that the package implements (example: Implementation of an interface).

### 3.3.1 Depends

Depends is used for packages that are required for the depending package to have a significant amount of functionality and/or for packages that are required for the postinst, prerm and/or postrm scripts in order for them to run properly. The depends-relationships are special because they can be conjunctive, meaning that several different packages can satisfy the same dependency.

### 3.3.2 Recommends

Recommends lists packages that under usual circumstances are found together with the package.

### 3.3.3 Suggests

Suggests lists packages that enhance the functionality of the original package.

### 3.3.4 Enhances

Works like suggests but works in the opposite direction.

### 3.3.5 Pre-depends

Works like *depends* but forces dkpg to install the packages which are declared in this list before the original package is installed.

### 3.3.6 Breaks

Breaks is used to declare that the package about to be installed breaks another package i.e. reveals a bug or takes over a file from an earlier version of the package. Breaks therefore usually has an “earlier versions than” clause.

### 3.3.7 Conflicts

*Conflicts* is a stronger restriction than breaks and is used when the conflicts are not resolved in later versions of the package/packages that are listed. It is also used in a construct where the package conflicts with a virtual package it is providing. This so that only one package providing a given service can be installed at a time.

### 3.3.8 Provides

Provides lists virtual packages which are provided by the package, mainly for constructs like the one mentioned in *Conflicts* above and other.

### 3.3.9 Replaces

Replaces has two different uses. Together with breaks, it enables the package to take over a file in another package which in later versions does not own the file in question. Together with conflicts it enables the package to fully replace another package and force its removal.

# 4 Sources

## 4.1 Supervisor at Sony Ericsson

The assigned supervisor for this bachelor thesis, Axel Bengtsson, Staff Engineer at SEMC Lund, has been a valuable asset in every aspect of this thesis.
The conclusions about APT-get’s solving algorithm, has in part been drawn from his experience from working with the issues of APT-get on a daily basis. Furthermore his knowledge about the organizational infrastructure surrounding the software composition proved valuable in determining vital stakeholders in the requirement elicitation for the requirement specification and laid down the groundwork for a deeper understanding of the origins of these requirements.

## 4.2 Mancoosi research project and partners

In the search for relevant literature on package dependencies, it became apparent that the bulk of research in package dependencies, with regards to open source systems (like Debian), originates from the Mancoosi research project and its predecessors and academic partners, the Edos project and Inria. This has led to that the main part of the theory on package dependency solving and various solving algorithms, in this thesis stems in part or wholly from this source. This can be perceived as problematic.
Since Mancoosi is a non-profit research organisation, their main interest lies in solving the problems and finding out the facts to the best of their abilities, and that the results are not, to any greater extent influenced by other factors. Since Mancoosi takes on such a leading role in this field of research it is difficult not to take their findings into account. With this in mind the fact that most of the literature and theory that this thesis relies on comes largely from one single source remains to be taken into consideration.

### 4.2.1 Inria

Inria stands for;“Institut National de Recherche en Informatique et en automatique” (National Institute for Research in Computer Science and Control). Created in 1967 Inria is a Public Scientific and Technical Research Establishment under the supervision of various French authorities.

### 4.2.2 The Edos project

The Edos project was the predecessor of the Mancoosi project and ran between 2004 and 2007 (Mancoosi). It was aimed at the stability of distributions and the development of tools for checking the consistency of a set of packages.

### 4.2.3 The Mancoosi project (Mancoosi)

Mancoosi is a European research project in the 7 th Research Framework Programme (FP7) of the European Commission. Mancoosi stands for

“MANaging the Complexity of the Open Source Infrastructure”. While the Edos Project was aimed at the distribution side of open source infrastructure, the Mancoosi project is aiming at developing tools for the system administrator.

## 4.3 The packages file

The packages file is a file that consists of the metadata for every package in a repository. It is used by APT get to find out which packages are available and the paths to the different packages files are found in the sources.list file.

# 5 APT-get

## 5.1 APT-get solving algorithm

Since part of the requests from SEMC was to find alternatives to APT-get, an important assignment would be to investigate APT-get's algorithm for solving package dependencies. However I have been able to find little to no documentation on how APT-get behaves "under the hood", part from innuendos and guesswork. Contacting the maintainer of the APT-get package led to nothing. The two possible alternatives left were reversed engineering or studying the source code. After talking with the skilled engineers and programmers at SEMC, who themselves tried to penetrate the 15+ years old tapestry that represents the APT-get source code, I concluded that it could not be done within the confinements of this bachelor thesis. The extensive research needed to conduct a "proper" reversed engineering, would also be difficult to conduct within the scope of this thesis. Therefore this chapter is merely a record of a perception of how apt-get's algorithm works, and is largely based on the findings of Mancoosi's research. It's an attempt to show some of the problems and shortcomings of APT-get.

### 5.1.1 Algorithm specification (EDOS project team, 2006)

This specification comes directly from the EDOS project work page, for a more in depth explanation of how they reached these results, consult the source.

1. Check the dependencies of a package.
2. Try to install dependencies one-by-one in the order they are presented in the meta-data.
3. For each dependency try to install its sub-dependencies by the greater version presented.
4. If one sub-dependency fails by conflict with a package that will be installed, then the install operation aborts. It does not try to back track and check smaller versions.
5. If one sub-dependency fails by conflict with a package that is already installed, then the install prompts for removal of the installed package. It does not try to find alternatives for the conflict package.

The specification was deducted from a simple test with a package base of 15 packages with an optimal solution. APT-get did not manage to find said solution.

### 5.1.2 Result

This shows how APT-get does not investigate every possible configuration but instead, when heuristics fails, prompts the user to change the installation manually to fit APT-gets needs. Therefore can be deduced, given the results from the EDOS project test results, that APT-get's solving algorithm is incomplete.

### 5.1.3 Conclusion

The algorithm used by APT-get to solve dependencies to enable package installment seems to be based on a semi-randomized trial-error methodology with some heuristics. To install a package A, APT-get inserts every node (package) that A depends on (and their dependencies, and their dependencies and so on…). Thereafter APT-get builds a graph and traverses this graph, trying to find a set of packages that solves the dependencies without being restricted by constraints (i.e. conflicts or breaks). Testing any possible combination of these packages is not reasonable for a large amount of packages; hence APT-get has a time/operation restriction. After trying a set number of combinations without finding such a set, APT-get concludes that there are no solutions that will enable the installment of package A. Furthermore if APT-get finds a set of packages that enables the installation of A, APT-get does not take into account if packages that are not necessary to install A is installed also, as long as they don't conflict with any other installed package. This leads to, at least two major issues:

- An existing solution is not found by APT-get.
    - That this behavior is unfortunate is easily recognized.
- One or several packages are installed without there being a need for it.
    - This behavior is not fatal on larger systems, for example a ubuntu system on a pc, in such a case the user would most likely never realize that additional packages had been installed, and since no installed package depends on them they would be scheduled for removal if later installation conflicts with them.
    - On a smaller system, like an android system on a mobile phone, extra packages use a larger percent of the recourses and there is a larger risk of unwanted behavior and legal issues (specially on a international product, such as a mobile phone, where some applications are not allowed in certain countries).

## 5.2 APT files

In the dependency graph visualization implementation some of the log files used by APT-get are used to gather information about the build. This is a short description of these files.

### 5.2.1 var/log/apt/history.log

A history log that registers which packages where installed.

### 5.2.2 etc/apt/sources.list

This file contains a list of paths, local and remote, used by APT-get to find the packages to install, i.e. the repositories.

## 5.3 Pinning (APT team, 2003)

To select which version of a package to install, APT-get sets the priority of each version of a package by giving it a “pin”-number. These pin-numbers can be assigned by default, through specifying them in the APT preferences file or specifying them with a command line argument to APT-get. The priorities are interpreted as follows:

- $\boldsymbol{P} > 1000$ – The package is installed even if it means a downgrade from the already installed package
- $\mathbf{990} < P \leq 1000$ **–** The package is installed even if it is not part of the “target release”, unless the installed version is more recent.
- $\mathbf{500} < P \leq 990$ – The package is installed unless there is a version available that belong to the “target release” or if there is a more recent package already installed.
- $\mathbf{100} < P \leq 500$ – The package is installed unless there is a more recent version available or already installed.
- $\mathbf{0} < P \leq 100$ - The package is installed only if there is no installed version of the package
- $\boldsymbol{P} < 0$ – The package is never installed

### 5.3.1 Default assignment

If no other pin is specified the package version receives the same priority as the distribution to which it belongs. There are two ways of making a specific distribution the “target release” (i.e. give it the highest priority) ; by specifying it in the APT configuration file (APT::Default-Release “stable”) or by specifying it in a command line argument (APT-get install –t testing <package name>). APT-get gives the packages three different, default, values:

- **100** – to versions already installed
- **500** – to versions who are not installed and don’t belong to the specified release, whether one is specified or not.
- **990** – to versions who are not installed and belong to the specified release, if one is specified.

### 5.3.2 APT preference file

To specify a pin-priority for a package or several packages you use the following notation:

- Specific package and version:

Package: <package name>
Pin: version <package version>
Pin-Priority: <priority>

- All packages from a specific location

Package: *
Pin: origin “<package producer>”
Pin-Priority: <priority>

- All packages from a specific release with a specific version

```
Package: *
Pin: release a=<archive name>, v=<version>
Pin-Priority: <priority>
```

# 6 Package dependency solving

## 6.1 Definitions (Treinen R. Z., 2008)

To simplify the reasoning we let $\pi$ be a package in a repository R, and set $\pi$ to be installed in an empty environment.

We define the function:

$$D(\pi) = \{\pi \cup D(\pi_1) \cup D(\pi_2) \cup \dots \cup D(\pi_n)\}$$

where $\pi_1, \pi_2, \dots, \pi_n$ are the packages that $\pi$ depends on directly. $D(\pi)$ can therefore be said to represent a set of packages with every package that $\pi$ is directly or indirectly depending on for its installation.
It is trivial to deduce that $D(\pi)$ contains at least one set of packages that, without regards to constraints, solves the dependencies of package $\pi$.
We can therefore state that $D(\pi)$ has a set of subsets $\Pi_1, \Pi_2, \dots, \Pi_i;\ 1 \leq i \leq n$ where each subset is enough to solve the dependencies of $\pi$. Given the repository $R$ as a set of packages $\{p_1, p_2, p_3, \dots, p_n\}$, we define the function, C(R), as the set of conflicts and/or other constraints $\{p_i, p_j\}$ that exists between the packages in R.

$$C(R) \subset R \times R$$
$$C(R) = \{(p_i, p_j) : p_i, p_j \in R \; and \; p_i \; conflicts \; p_j\}$$

**Definition** For package $\pi \in R$ to be installable with regards to R, $D(\pi)$ has to have at least one subset: $\Pi_i \subset R$ where $(\Pi_i \times \Pi_i) \cap C(R) = \emptyset$.

## 6.2 Heuristic rules

- The relationship “predepends” is a stricter version of depends, and can therefore be treated as a “depends” relationship when resolving the dependencies of a package.
- The relationships “recommends”, “suggests” and “enhances” are not vital to resolving the package dependencies.
- The relationships “breaks” and “conflicts” may restrict the ability of one or many “sub-dependencies” to be resolved.
- If there are several different alternative sets of packages that will resolve a certain package dependency, and at least one of those “sub-dependencies” is resolved, the inability to resolve the other, alternative, sets does not affect the possibility to resolve the entire dependency-tree.
- The relationship “replaces” is just a marker for special cases of “conflicts” or “breaks”, and as such it does not, in itself, affect the ability to resolve the package dependencies.

## 6.3 Package pendency solving as a SAT[1] problem (Treinen R. Z., 2008)

The lack of completeness in APT-get’s solving algorithm, discussed above, is not a problem for the average Debian user, who will most likely not even notice that his system contains packages that are not used by the other parts of the system. They shouldn’t cause any major conflicts and if they do some time in the future they are bound to be removed, since there are no other packages that depend on them. But for a smaller system like an android system on a smart-phone, extra packages without any function are using valuable resources and are most unwelcome.
When treating dependency solving as a SAT problem end reverting the problem into solving a Boolean expression you can create a solver that is complete. And furthermore if reverting it to a pseudo-Boolean problem you can optimize the solution with regards to different variables: size, latest version etc.

### 6.3.1 Expansion

Before reverting the dependencies into a Boolean expression one must first expand the expressions so that expressions of the type,

```
package: a
depends: b (>= 2), d
conflicts: c
```

,given that package b exists in version 1, 2 and 3, becomes the discrete expression:

```
package: a
depends: b(=2) | b(=3), d
conflicts: c
```

Furthermore the virtual packages have to be reverted into Boolean expressions as well. This is done by simply stating that the virtual package is a package that depends on the various packages that provides it,

```
package: a
…
provides: v

package: b
provides: v
```

[1] Boolean satisfiability problem

which gives the virtual package v the following expression:

package: v
depends: a | b

The rest of the packages in this short example, introducing the dependencies of the two separate versions of package b, and the two packages d and f that have no dependencies, would then be expanded as follows:

package: a
depends: b(=2) | b(=3), d
conflicts: c

package: b
version: 2
depends: d, f
conflicts: e

package: b
version: 3

package: d

package: f

### 6.3.2 Installabilty of a package as a Boolean expression

**Definition 1** An installed package corresponds to the Boolean value “true” whereas a package wich is not installed is corresponds to the Boolean value “false”.

**Definition 2** B(p) denotes the Boolean expression of the installability of package p.

The Boolean expression for package a and the packages a depends upon or conflicts with, in the previous example would then be:

$$B(a) = ((b_2 \lor b_3) \land d \ \land \bar{c})$$
$$B(b_2) = (f \land d \ \land \bar{e})$$
$$B(b_3) = (b_3)$$
$$B(d) = (d)$$
$$B(f) = (f)$$

This interprets as: For the expression B(a) to result as a “true”, package d has to be installed with either package b version 2 or package b version 3, package c must not be installed.

### 6.3.3 Boolean expression for an entire installation

The Boolean expression for the installation of package a in the above example would be:

$$B(a) \wedge (B(b_2) \vee B(b_3)) \wedge B(d))$$

This has two solutions:

1. a = true, $b_2$ = true, d = true, f = true, c = false, e = false (with package $b_2$ installed)
2. a = true, $b_3$ = true, d = true, c = false (with package $b_3$ installed)

### 6.3.4 Optimization

A simple example is optimization with regards to number of packages, which would translate to the solution with the fewest amounts of “true” variables (installed packages). As not to get close to infinite number of solutions, one must first discard the trivial solutions where packages that does not depend on or conflict with any other packages and are not depended on by any of the packages in the installation are installed.
Using a pseudo Boolean expression where “false” is interpreted as 0 and “true” is interpreted as 1, and the packages are weighted with one of their attributes: size, version, accessibility etc, makes it fairly straightforward to choose the installation that requires the least disc-space or which installation that takes the least time to download. An example of optimization with regards to memory would be the given example of an installation of package a (above) where the packages would occupy memory as follows:
Package a – 1kB
Package $b_2$ – 2kB
Package $b_3$ – 4kB
Package d and f – 0.5kB
Solution 1: $1kB * a + 2kB * b_2 + 0.5kB * d + 0.5kB * f = 3kB$
Solution 2: $1kB * a + 4kB * b_3 + 0.5kB * d = 4.5kB$

Such optimization might come in handy in the general case of package installation, whereas in the case of SEMC’s package management, multiple possible solution, would translate into a non valid installation since specific software configured according to customer requirements cannot have any alternatives.

# 7 Using APT-get with external solvers

## 7.1 Technical background

### 7.1.1 APT-CUDF (Debian team)

APT-cudf provides integration between APT-get (EDSP) and external, CUDF-based, dependency solvers.

### 7.1.2 CUDF (Treinen & Zacchiroli, 2009)

CUDF, Common Upgradeability Description Format, is developed by Mancoosi and is used to describe and encode upgrade scenarios. It is an intermediary file format between APT-cudf and the external solver.

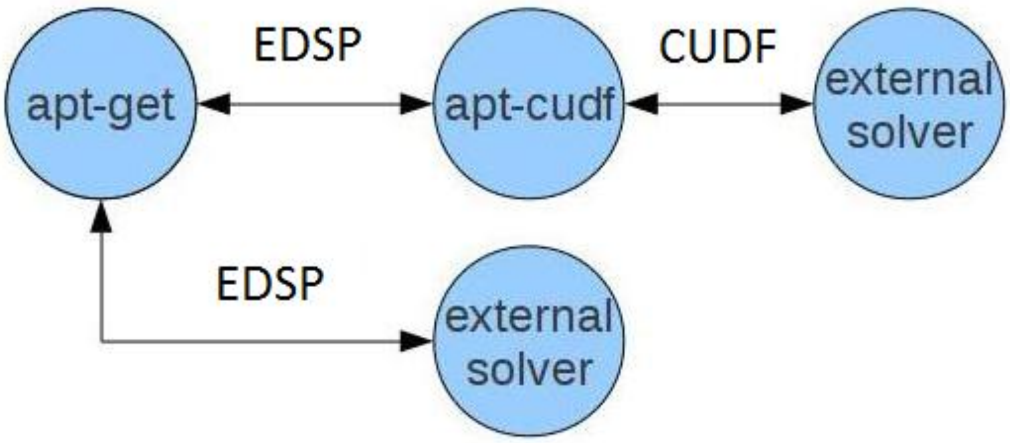
### 7.1.3 EDSP

EDSP is an intermediate file format between APT-get and external solvers. It resembles the Packages file (described in 2.3) and CUDF but has some APT-specific fields. This intermediary format can be used to link an external solver to APT-get. It is supported from APT version 0.8.16, which can be found in debian-experimental and ubuntu-oneric amongst other distributions.

## 7.2 External solvers

With the aforementioned framework there are two possible alternatives:

1. Connect the external solver at the EDSP level.
2. Use APT-cudf and connect the external solver at the CUDF level.



Picture 1 Schema over APT-get with external solvers.

## 7.3 Finding alternatives to APT-get, the SEMC Solver

One of the objectives of this thesis was to find and evaluate alternatives to APT-get for SEMC. The findings presented in this thesis, with regards to using APT-get with an external solver, was regarded as an optimal solution, since there would be no need to change meta installer, with the overhead that this would result in.

The first attempt with a SAT based solver gave very promising results as this solver managed to pass all the scenarios considered to mimic the complexities in the near future.
Shortly thereafter SEMC started to develop their own solver that would be tailored for their needs and the task of finding alternatives was deemed completed.

# 8 Graph representation of package Dependencies

In the graph representation of package dependencies the actual graph consists of the “depends” relationships that one package has to other packages. The other relationships are regarded as attributes of the various nodes.

## 8.1 Terminology

A short summary, explaining the different terms used in this thesis when discussing dependency graphs:

**Root** – Representing the product package, or the “top” of the dependency graph.
**Node** – Representing a package in the dependency-graph structure. Root is a special case of node.
**Edge** – In the graph representation of the dependency structure a dependency is represented by an edge.
**Sub-dependency-graph** – A dependency graph that begins with a node, in the dependency graph, other than the root.
**Resolved dependency** – a dependency where each sub-dependency is resolved.
**Dependency-graph** – Representing every package and dependency from the root down, resolved or unresolved.
**Resolved dependency-graph** – A dependency graph with every dependency resolved from the root down.

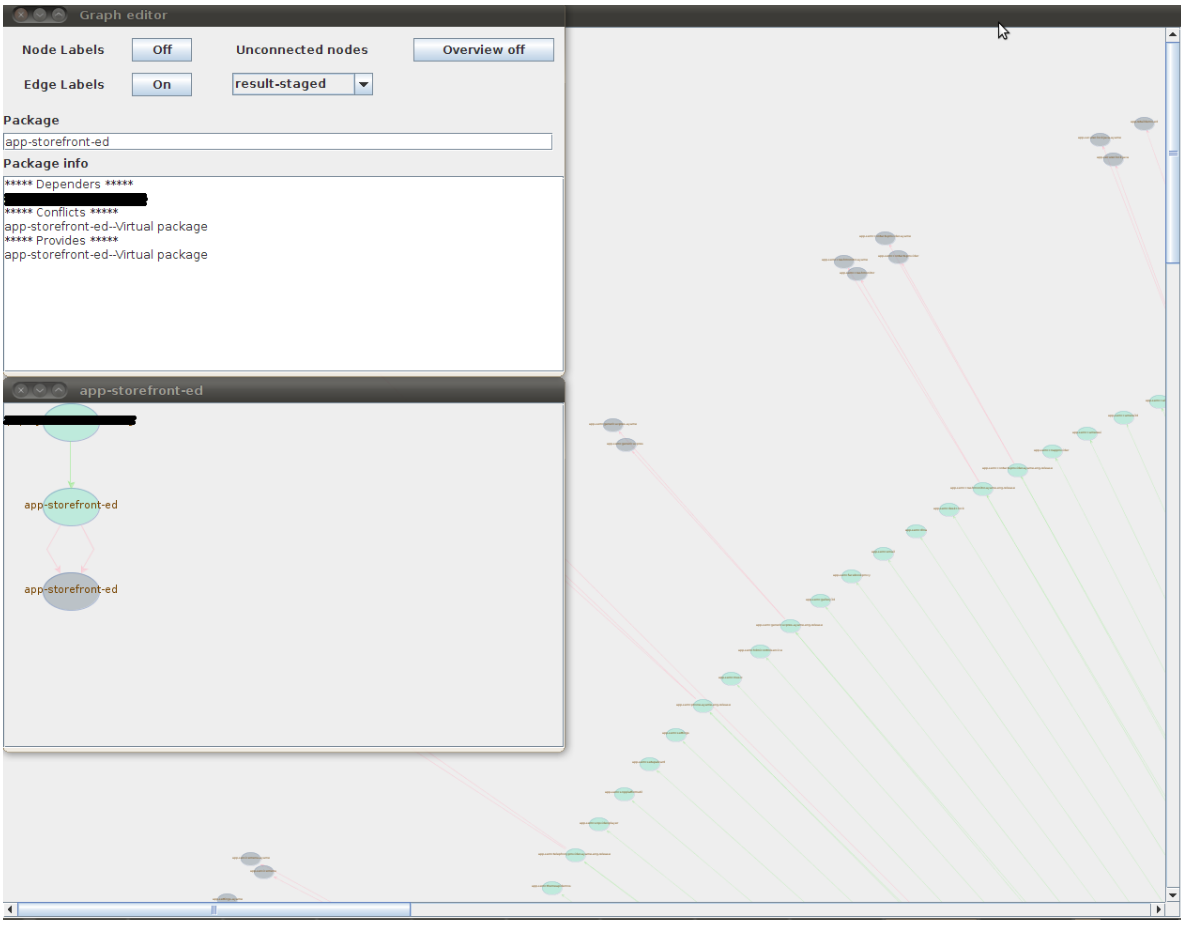
## 8.2 Circular dependencies

A circular dependency is a dependency where a package depends on another package that directly or indirectly depends on the first package. Such a construction is possible and supported by APT-get. It is however considered a bad construct. With packages with circular dependency it is obvious that at least one of the packages will fail to have its dependencies resolved before configuration.

### 8.2.1 Dealing with cyclic tendencies in a DAG (Directed Acyclic Graph) representation

In this thesis I will not attempt to solve these cyclic tendencies, with respect to which package should be unpacked first. In that sense cyclic dependencies are not a problem I will consider as such. However with the presence of cyclic dependencies it is no longer certain that the graph representing the dependencies is a DAG.
To prevent this scenario I will treat the loop back to the previous package as a “virtual” sub-graph that is resolved when/if all other dependencies are resolved for the two inter-depending packages.

# 9 Pamp – A Package Dependency Graph Visualization tool



Package
app-storefront-ed
Package info
```
***** Dependers *****
[redacted]
***** Conflicts *****
app-storefront-ed--Virtual package
***** Provides *****
app-storefront-ed--Virtual package
```
app-storefront-ed
[redacted]
app-storefront-ed
app-storefront-ed

Picture 2 The main menu of Pamp, the node ”app-storefront-ed” is selected in the overview graph and its immediate relationships are described in text in the “package info” window, and in a graphical presentation below the main menu. Traversal is made possible by selecting nodes in the sub-graph-window to make them come in focus.

## 9.1 Introduction

One part of this bachelor thesis was to create a Visualization tool for Debian packages in a SEMC software composition. As a request from SEMC this tool was implemented using the Java programming language.

### 9.1.1 Technologies

Programming language: Java SE 1.6
External libraries: JGraphX Version 1.8.0.2 – Graph visualization library.
IDE: Eclipse Indigo Version 3.7.0

## 9.2 Start

### 9.2.1 Input Parameters

- Name of the temp catalog.
- Variant – SEMC specific identifier for different variants of a composition.

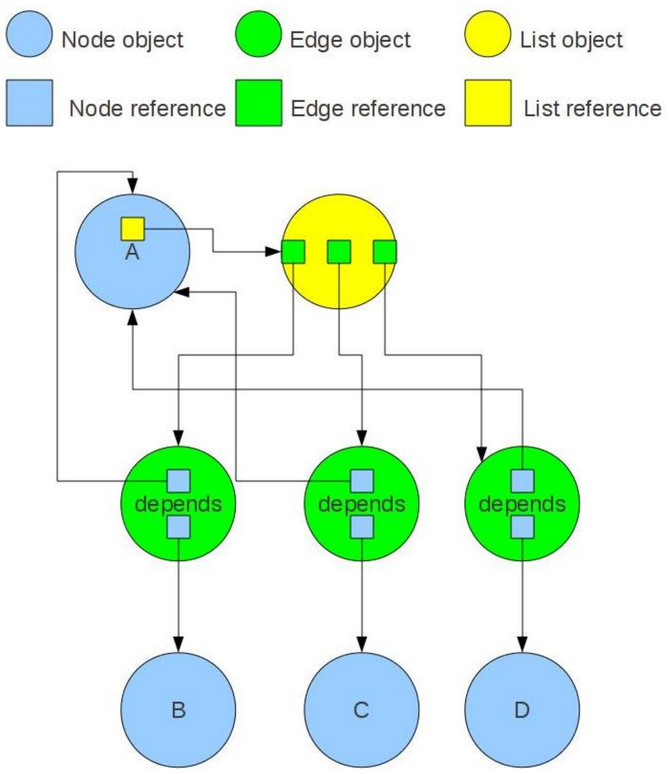
- Root package – The name of the “main” package.

### 9.2.2 Set Up

When building a software composition the actual build is made in a temp catalog in which an environment, for APT-get to function in, has been set up. Given the name of this catalog the etc/apt/sources.list file in said catalog is parsed to get the packages-files of the used repositories. From these repositories meta-data about the packages are gathered.
From the temp catalog /var/log/apt/history.log is parsed to get which packages were installed. If no packages were installed then the packages that should have been installed, did the build succeed, are used instead.

## 9.3 Data objects and data object construction.

In accordance with the theory of chapter 3 the graph consists of nodes and edges.



Picture 3 Description of a simple depends relationship described with the data objects, package: A, depends: B,CD.

### 9.3.1 DependencyNode

The DependencyNode has the following attributes.

- Package name – The name used in the Package meta data.
- Package ID name – Node specific name (a concrete and a virtual can share the same name, there can be several versions of the same package with the same name).
- Version – the version of the package.
- Package status – Concrete, Virtual or Broken (a package is considered to be broken if another package is depending on the package and the

package is not available from the repositories specified in the temp catalog).
- Lists of DependencyEdges representing the different relationships specified by the Debian Binary Package Control File dependency fields, as specified in 2.3.

The construct of having the DependencyEdges in lists enables easy traversing of the graphs.

### 9.3.2 DependencyEdge

Given the nature of the depends-relationship (see 2.3.1) the dependency edges come in two flavors.

- DependencyEdge
- MultipleDependencyEdge (that inherits DependencyEdge)

With this construction all DependencyEdges can be treated the same way dynamically. When installed the MultipleDependencyEdge works as a DependencyEdge since the dependency is solved by only one of the possible alternatives.
The DependencyEdge has the following attributes:

- Multiple flag – a flag representing whether the DependencyEdge is multiple or not.
- Resolved flag – a flag representing whether the DependencyEdge is resolved or not. Used if all the alternatives are to be displayed in the graph.
- Origin DependencyNode – a reference to the DependencyNode where the DependencyEdge originates.
- Endpoint DependencyNode – a reference to the DependencyNode that is the DependencyEdge's endpoint.
- Endpoint name – used to get the reference of the endpoint DependencyNode, when a DependencyEdge is created only the origin is known, the endpoint is solved later.
- Edge type – the type of relationship this DependencyEdge represents (see 3.2.1).

The MultipleDependencyEdge also has the attribute:

- Alternatives – a list of DependencyEdge's representing the different alternatives.

### 9.3.3 DependencyNodeFactory

A factory that creates DependencyNodes from the metadata provided in the set up.

## 9.4 Building the Graph

After the set-up, all the data that is needed to building the package dependency graph is available.

### 9.4.1 PackageDepenencyTreeBuilder

First of the graph is built recursively from the root-package and down with the PackageDependencyTreeBuilder using the nodes provided by the DependencyNodeFactory. The PackageDependencyTreeBuilder also holds the attribute:

- Installed packages – a HashMap of the packages that where installed (if any).

### 9.4.2 PackageDependencyTree

(The name “tree” is somewhat misguiding because the graph in question is not necessarily a tree. The term “tree” is therefore not equivalent of the graph-theory term.)
The result of the graph build is a PackageDependencyTree that have the following attributes:

- Nodes – A HashMap of the nodes of the tree.
- Edges – A HashMap of the edges of the tree.
- Root package – the name of the package that is the root of the tree.
- Solved flag – a flag representing whether the graph is solved.

The PackageDependencyTree also has functionality to:

- Create a PackageDependencyTree that represents a sub graph represented by the specified DependencyNode and its immediate “neighbors” (This sub-tree is used to implement the graph traversing functionality).
- Return a list with packages that has a relationship to the given package and the nature of those relationships (see 2.3.1).

## 9.5 Visualizing the graph

The classes used for visualization of the graph where based on the JGraphX library.

### 9.5.1 GraphVisualizer

An abstract class was created to deal with common functionality such as node and edge shape and color, producing visual nodes and edges from the edges and nodes in a PackageDependencyTree, various settings for layouts etc. Furthermore this class contained functionality to alter the visualized graph: set visual edge and/or node labels etc.
The class was instantiated by two concrete classes:

- One to display an overview of the SEMC specific dependency graph.
- One to display a sub graph, which was used when traversing the graph.

## 9.6 JGraphX

JGraphX is a library based on the Swing framework. Its main use is for graph presentation, normally with much smaller graphs than the ones I aimed to use it for.

#### 9.6.1 Selecting graph library

After evaluating different graph visualizing libraries such as Graphviz and Jung, the decision was made to go with JGraphX.
The main reasons for choosing JGraphX where:

- Java library – no porting
- Free/Open source
- Swing compatible – easy to integrate into a Swing GUI
- Flexible
- Up-to-date

Initial trials proved that it would be possible to produce an acceptable result with this library.

#### 9.6.2 Working with JGraphX

As JGraphX is a free Java version of a non-free javascript library most of the tutorials and support forums concerned the javascript version, which naturally have a slightly different behavior. This together with the fact that the library is poorly documented, if documented at all, resulted in that the workload to "get to grips" with the library was quite extensive; two to three working weeks. Nevertheless after it was tweeked into functioning as expected there was much to be gained by using functionality that was already built in.
Even though it meant some hacking to modify it to function for larger graphs, the result was pleasing and the choice of graph library was, if not optimal, than by far satisfactory.

### 9.7 Overview layout

Since graph visualization is regarded as a NP complete problem, making a visualization tools for visualizing an overview of a arbitrary graph would fall way out of the scope for a bachelor thesis, even creating a layout that would present a readable graph for the specific graph provided by SEMC's composition proved to be challenging. Since the graph was fairly flat, quickly expanding in the early stages, the only way to present it would be to place the root node in the middle of a circular layout of the nodes in the next stage, thereafter the sub graphs of each node in this circle would expand perpendicular from the circles edge, creating a sun like pattern.
Implementing this with the JGraphX library was no easy task, and this part of the implementation was by far the most timeconsuming.

### 9.8 Graph traversing

The graph traversing was implemented by displaying the chosen node in a small sub-graph with all the nodes that have direct relationships with the selected node. In this graph every node is selectable and on selection the sub-graph for the selected node is displayed. This, together with a text field that displays the meta- data concerning the selected node, proved to be a powerful tool for graph traversal. In most cases an overview says little about the problem at hand. Focusing on the area around the problem-package gives more information about the problem at hand.

### 9.8.1 Why graph visualization and traversing?

The most common way of debugging a faulty installation is by examining log-files or the packages-file itself. The latter means following the relationships in a file that lists 8000+ packages, searching for anything in such a file, with only the “find” command of the text editor, is a fairly time consuming task. Man is not optimized for searching lengthy text-files; a graphical representation makes more sense and is more easily comprehended.

# 10 Package Dependency Requirements Specification

The requirements specification consists in large parts of information that SEMC, of obvious reasons, does not wish to be displayed in an official document. This chapter is therefore a general description of the requirements and the elicitation process.

## 10.1 Analysis

Since the goal of the requirement specification was to specify requirements on future dependency structure, it was vital to keep the requirements on an abstraction level above the present environment (i.e. APT-get and the Debian package). This also made it natural to keep the elicitation process as open discussions. To create a valid requirements-specification of dependencies, as used in the composition process, the first objective was to obtain a deeper understanding of the process that led to the dependencies being introduced and identify possible stakeholders.

## 10.2 Stakeholders

After analyzing the package dependencies, the processes that introduced them and the processes that dealt with them the following stakeholders became apparent.

### 10.2.1 Product Configuration Manager

Responsible for the package configuration top entities, and with the dependencies of these entities define the package configuration.

### 10.2.2 Software Configuration Manager

Responsible for compiling source code and package the resulting binaries in Debian packages and making them available in the repository. Also responsible for updating the packages-file of the repository.

### 10.2.3 External Applications Configuration Manager

Responsible for packaging external applications into Debian packages and making them available in the repository.

### 10.2.4 Platform Development Configuration Manager

Responsible for handling of platform specific resources. None of the requirements originated from Platform Development.

### 10.2.5 Content and Customization

Responsible for handling customer requests and providing a customization entity that implement said requests and making it available in the repository.

## 10.3 Start up

After defining the stakeholders and their role in the organization a meeting with these stakeholders was held to get a clearer view of what types of requirements were to be expected and which the common requirements where for the stakeholders. Instead of having a clear agenda, a graph of the workflow in direct relation to the dependencies where used as a discussion basis. The workflow graph consisted of the aforementioned stakeholders and the way they interacted with various components such as: the Debian package repository, the source code, customer requests etc.

## 10.4 The Document

After the startup meeting, when the nature of the requirements became more apparent I decided to compose the document as follows:

1. Introduction – Background and goal for the document as well as a short description of the content.
2. Stakeholders – A description and categorization of the stakeholders.
3. Elicitation – A record and short description of the various meetings, what subjects that where discussed and the requirements introduced at the meeting, if any.
4. Requirements – The requirements, categorized by the different stakeholders and one category for common requirements.
5. Requests – Requirements that were not directly related to package dependencies.
6. Issues – Issues with the composition that became apparent as the document progressed.

## 10.5 Elicitation meetings

The meetings were held at SEMC. After the start up meeting, meetings were held with the various stakeholders, department by department. When adjoining topics emerged meetings were scheduled with all the interested parties. The requirements and discussion from the start up meeting worked as a basis for discussion in the various follow up meetings.
Apart from being vital in requirement elicitation, these meetings also led to a deeper understanding of the background of the requirements and as a consequence led to a more concise formulation of the requirements.

## 10.6 The requirements

Apart from requirements specific for the SEMC software composition there emerged three common requirements:

1. Separate the “actual” dependency relationships from the structural definitions.
   1.1. The present package structure is used for structural purposes , these structural definitions could be handled at a different level then the package dependencies.

2. Let the *.deb package dependency attributes reflect the actual dependencies and conflicts of the resources contained within the package.
   2.1. The inter package relationships are at a abstraction level where the underlying dependencies and conflicts of the resources contained within the packages are not visible.
3. Change the structure to facilitate the possibility of defining higher level entities.
   3.1. If high level entities, such as features etc., where used, the dependency structure would to a higher extent mirror the underlying software.

## 10.7 An implementation suggestion

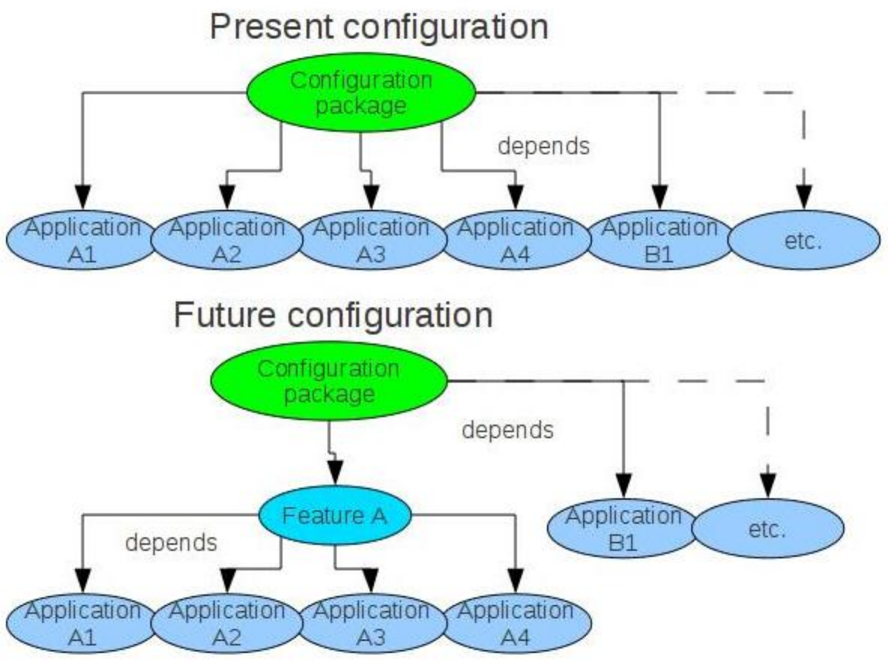
At request from SEMC, an appendix was added to the requirements specification. This appendix contained suggestions on how to implement the requirements specification. The implementation suggestion concerned mainly the issues presented in the first and third common requirements described in chapter 10.6. To keep track of the dependencies of the resources of every package was deemed a far to labor intensive task to fit the scope of a bachelor thesis, and therefore there was given no suggestion on how to implement requirements linked to this requirement.

### 10.7.1 First common requirement

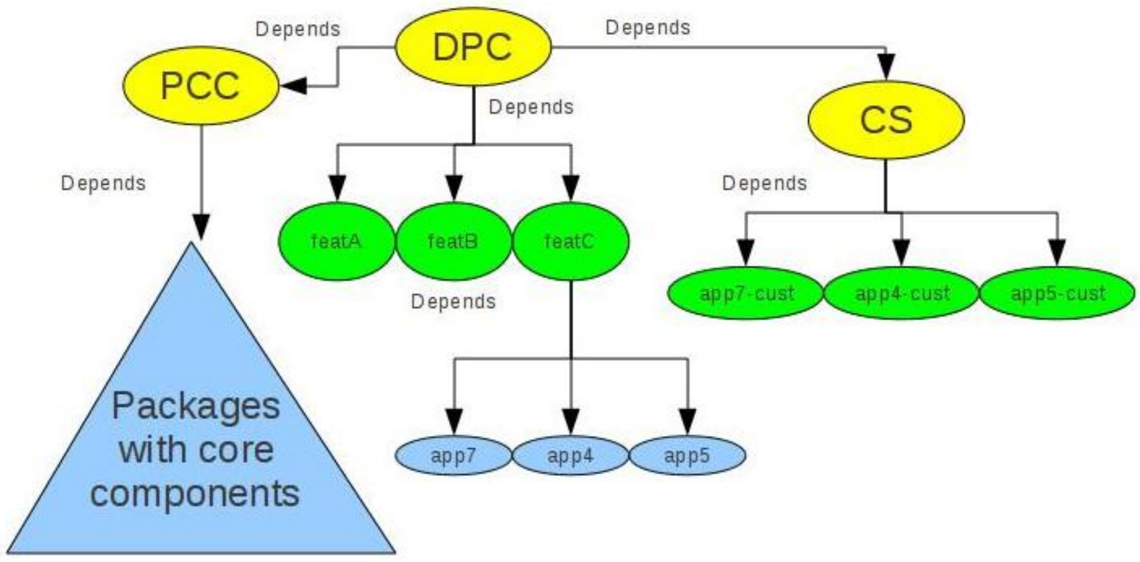
The implementation of the requirements related to the first common requirement mainly concerned different levels where to introduce the structural definitions. One possible alternative is the extraction of sub-packages-files at different levels: per product, per composition etc. one of the benefits from extracting a sub-packages-file for every composition would be the improved backwards-traceability of every build.
The extraction of these packages files could either take place pre packages file creation, from the data that is used to create the master packages file, or they could be parsed using a SEMC specific field in the package metadata.

### 10.7.2 Third common requirements

The implementations of the third, common, requirement, dealt with: how a dependency structure could be realized in order to implement the requirements concerning the ability to define features and other subsets of packages. One of the main benefits from such a structure would be that decisions on the top level configuration would be restricted to which features and functionality to include. The dependency structure would also be human readable since the dependencies would make the different high level entities identifiable. With the present dependency structure there is no easy way identify which packages that provides a certain feature, and removal or management of such a feature can be a difficult and time-consuming task.



Picture 3 Feature package configuration



Picture 4 Final configuration after implementation (DPC - Deliverable Product Configuration, PCC - Product Core Configuration, CS - Configuration Settings)

## 10.8 Positive side-effects

Apart from resulting in a requirements specification, the elicitation process led to a number of fruitful cross apartment discussion that in them self could prove to be vital in the composition structure remodeling process.

# 11 Conclusions

This thesis has successfully reached all of the three major goals described in chapter 1.4.

## 11.1 Implement visualization tool

All the requirements in the problem description have been fulfilled by the implementation, Pamp. Pamp have the potential of becoming an important tool for debugging dependency structures, since graphical representation makes the dependency structures more readable to man, and the traversing functionality increases traceability of dependencies.

## 11.2 Identify and specify requirements on dependency structures

The requirement specification together with the implementation suggestion gives valuable information about how the dependency structures of SEMC's build system could or should be modified, and is a vital piece in the puzzle towards modifying the product configuration process.

## 11.3 Find alternative solutions to present build system

APT-get, using its own dependency solving algorithm, does not fulfill the requirements SEMC has on a dependency solving meta-installer. APT-gets inability to find existing solutions and tendency to install packages without valid reason makes it a liability in the software composition process.

As presented in this thesis, using APT-get with an external solver, based on the Boolean method that provides complete solutions and possibility for optimization, is a powerful alternative that leaves customization of the other tools in the composition chain to a minimum since APT-get remains the interface towards package installation.

# 12 Future work

## 12.1 Further development of Pamp

The need for dependency graph visualization tools outside SEMC build system would promote developing Pamp into an open source application suitable for Debian package infrastructure in more general environments, i.e. Debian distributions on personal computers.

## 12.2 Graph visualization and graph layout

Further research into graph visualization in general and graph layout in particular should be promoted since graphs are a powerful tool to describe processes, not only in computer science and software installation, and visualization is vital in any information relay.

## 12.3 At Sony Ericsson Mobile Communication

Implementing some or all of the requirements in the requirement specification should be a main concern for SEMC, since they stem from internal, valid, sources. SEMC is currently developing their own solver based on the Boolean method, as a direct result of the findings made in this thesis described in chapter 6.3 and 7.

## 13 References

APT team. (2003, 8 17). *apt_preferences(5).* Retrieved 10 5, 2011, from http://linux.die.net/man/5/apt_preferences

Debian team. (n.d.). *Package: apt-cudf (2.9.8-1).* Retrieved 1 12, 2012, from packages.debian.org: http://packages.debian.org/experimental/apt-cudf

Debian Team. (2011, 08 27). *The Debian GNU/Linux FAQ: Chapter 8 - The Debian package management tools.* Retrieved 9 1, 2011, from www.debian.org: http://www.debian.org/doc/manuals/apt-howto

EDOS project team. (2006, 03 2). *EDOS - package managers.* Retrieved 01 13, 2012, from www.mancoosi.org: http://www.mancoosi.org/edos/manager/#toc30

Garbee, B. e. (2011, 12 31). *A Brief History of Debian.* Retrieved 1 11, 2012, from www.debian.org: http://www.debian.org/doc/manuals/project-history/index.en.html

Jackson, I., & Schwartz, C. (2011, 04 07). *Debian Policy Manual version 3.9.2.0.* Retrieved 01 11, 2012, from www.debian.org: http://www.debian.org/doc/debian-policy/

Joseph Diaz, J. P. (2000, 10 17). *A Survey of Graph Layout Problems.* Retrieved 01 19, 2012, from www.citeseerx.ist.psu.edu: http://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.12.4358&rep=rep1&type=pdf

Mancoosi. (n.d.). *EDOS.* Retrieved 01 17, 2012, from www.mancoosi.org: http://www.mancoosi.org/edos

Mancoosi. (n.d.). *mancoosi - managing software complexity.* Retrieved 01 17, 2012, from www.mancoosi.org: http://www.mancoosi.org

Stallman, R. (2012, 01 02). *Linux and the GNU Project.* Retrieved 01 11, 2012, from GNU operating system: http://www.gnu.org/gnu/linux-and-gnu.html

Treinen, R. Z. (2008, 11 24). *Solving Package Dependencies:.* Retrieved 09 16, 2011, from http://arxiv.org/abs/0811.3620v1

Treinen, R., & Zacchiroli. (2009, 11 24). *CUDF 2.0 specification.* Retrieved 01 12, 2012, from www.mancoosi.org: http://www.mancoosi.org/reports/tr3.pdf

# 14 Acronyms

| | |
|---|---|
| **SEMC** | **S**ony **E**ricsson **M**obile **C**ommunication |
| **APT** | **A**dvanced **P**ackaging **T**ool |
| **EDSP** | **E**xternal **D**ependency **S**olver **P**rotocol |
| **DPKG** | **D**ebian **P**ac**k**a**g**e, software used to install, remove and provide information about Debian packages. |
| **PAMP** | **Pa**ckage **M**anagement **P**roject |
| **DAG** | **D**irected **A**cyclic **G**raph |
| **SAT** | Boolean **Sat**isfiability |
| **CUDF** | **C**ommon **U**pgradability **D**escription **F**ormat |