Eindhoven University of Technology

MASTER

# Linux package dependency visualization

Ernest Mithun, X.L.

*Award date:*
2011

Link to publication

**Disclaimer**
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

**General rights**
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

EINDHOVEN UNIVERSITY OF TECHNOLOGY
Department of Mathematics and Computer Science

# Linux Package Dependency Visualization

Xavier Lobo Ernest Mithun
(0666495)
Master's Thesis

Supervisors:
dr.ir. H.M.M. van de Wetering
ir. H.T.G Weffers, PDEng

Eindhoven, The Netherlands
August 2009

# Contents

LIST OF TABLES .......... IV

LIST OF FIGURES .......... IV

ACKNOWLEDGEMENTS .......... VII

ABSTRACT .......... IX

DEFINITIONS .......... X

INTRODUCTION .......... 11

1.1 Package .......... 11
1.2 Package Management Software .......... 12
1.3 User Questions .......... 13
1.4 Requirements .......... 14
1.5 Report Outline .......... 14

RPM PACKAGE DATASET .......... 15

2.1 Red hat Package Manager (RPM) .......... 15
2.2 RPM Design Goals .......... 15
2.3 Structure of an RPM Package .......... 16
2.4 Retrieving the RPM Package Data .......... 18

THE VISUALIZATION TOOL .......... 23

3.1 Abstract Data .......... 23
3.2 Force Directed Layout .......... 23
3.3 Filtering .......... 24
3.4 User Interaction .......... 25
3.5 Tool Features .......... 25
3.6 PREFUSE Visualization Toolkit .......... 36
3.7 Applications of the Visualization Tool .......... 37
3.8 Limitations of the Visualization Tool .......... 37
3.9 Possible Solution .......... 37

CIRCULAR LAYOUTS .......... 39

4.1 Circular Layouts .......... 39
4.2 Improving Circular Layouts .......... 40
4.3 Node Ordering .......... 42
4.4 Edge Bundling .......... 52

EXPERIMENTAL RESULTS .......... 55

**CONCLUSIONS AND FUTURE WORK** ........ **61**
6.1 CONCLUSIONS ........ 61
6.2 FUTURE WORK ........ 62
**REFERENCES** ........ **63**

# LIST OF TABLES

TABLE 1: COMPARISON OF NODE ORDERING TECHNIQUE........ 56

# LIST OF FIGURES

FIGURE 1.1 INSTALLATION OF A SOFTWARE USING PMS. ........ 13
FIGURE 2.1 STRUCTURE OF AN RPM PACKAGE ........ 16
FIGURE 2.2 PACKAGE A REQUIRES A CAPABILITY THAT PACKAGE B PROVIDES ........ 18
FIGURE 2.3 XML TO GRAPHML CONVERSION ........ 19
FIGURE 2.4 XML PACKAGE INFORMATION STRUCTURE. ........ 20
FIGURE 2.5 AN EXAMPLE DIRECTED GRAPH FOR RPM INFORMATION. ........ 21
FIGURE 2.6 GRAPHML PACKAGE INFORMATION STRUCTURE ........ 21
FIGURE 3.1 GRAPH DRAWN USING FORCE DIRECTED LAYOUT ........ 24
FIGURE 3.2 NODE HIERARCHY. ........ 25
FIGURE 3.3 SUB-GROUPS OF THE DEVELOPMENT GROUP ........ 26
FIGURE 3.4 MEMBER PACKAGES WITHIN THE DEVELOPMENT/DEBUGGERS SUBGROUP. ........ 27
FIGURE 3.5 PACKAGE CAPABILITIES ........ 28
FIGURE 3.6 PACKAGE RELATIONSHIPS ........ 29
FIGURE 3.7 SUB-PACKAGE HIDING ........ 30
FIGURE 3.8 PACKAGE HIDING ........ 31
FIGURE 3.9 THE FORCE DIRECTED VISUALIZATION WHEN GLIBC IS HIDDEN ........ 32
FIGURE 3.10 BACKUPPC PACKAGE REMOVAL ........ 33
FIGURE 3.11 BACKUPPC PACKAGE REMOVAL PATH ........ 34
FIGURE 3.12 IMPACT OF REMOVING THE HTTPD PACKAGE ........ 34
FIGURE 3.13 BACKUPPC - PERL PACKAGE RELATIONSHIP ........ 35
FIGURE 3.14 IMPACT OF REMOVING THE HTTPD PACKAGE ........ 35
FIGURE 3.15 [1] THE PREFUSE VISUALIZATION FRAMEWORK ........ 36
FIGURE 4.1 CIRCULAR LAYOUT OF A RANDOM GRAPH [9] ........ 40
FIGURE 4.2 BALLOON TREE LAYOUT ........ 41
FIGURE 4.3 EXTRAVIS CIRCULAR VIEW ........ 41
FIGURE 4.4 CIRCULAR GRAPH IMPROVEMENTS ........ 42
FIGURE 4.5 A LINEAR ARRANGEMENT OF NODES ........ 43
FIGURE 4.6 DIVISION OF NODES ON A CIRCLE ........ 45
FIGURE 4.7 NODE ORDERING OF $\bar{V}$ ........ 46
FIGURE 4.8 MINCA_DP NODE ORDERING ........ 50

FIGURE 4.9 HIERARCHICAL NODE ORDERING PROBLEM ........ 51
FIGURE 4.10 EDGE BUNDLING ........ 53
FIGURE 5.1 NODE ORDERING VISUALIZATION COMPARISON ........ 56
FIGURE 5.2 ZOOMED NODE ORDERING COMPARISON ........ 57
FIGURE 5.3 HIERARCHICAL NODE ORDERING ........ 58
FIGURE 5.4 ZOOMED HIERARCHICAL NODE ORDERING ........ 58
FIGURE 5.5 EDGE BUNDLING BENEFITS ........ 59
FIGURE 5.6 EDGE BUNDLING WITH NODE ORDERING ........ 60

# Acknowledgements

I would like to express my sincere gratitude to my project advisor dr.ir. H.M.M. van de Wetering for his continuous support and guidance throughout the duration of my thesis project and report writing. Without his encouragement and suggestions, this project might not have seen its completion. I am very grateful to my co-advisor ir. H.T.G Weffers, PDEng for his valued inputs and advices throughout the project. I am also thankful for his ideas on report writing and presentation. I would also like to thank dr.ir. D.H.R. Holten, for providing me all the necessary specifications regarding his work in a similar domain. I am thankful for dr. Serguei Roubtsov for his opinion on the importance of my work in industry. I am greatly indebted to dr. M. M. Pai and dr. R. M. Pai for considering me worthy for this great opportunity to gain and expand my knowledge through this program. I thank Mr. Balachandra for evaluating my thesis report. Finally, I would like to thank my family, friends and colleagues for their moral support.

Xavier Lobo Ernest Mithun
Eindhoven
25th August 2009

# Abstract

*In UNIX-like operating systems such as Linux, the software is split into thousands of packages which are tracked by package management software. Interdependency is an important term in this context because in a Linux system, the working of an application may depend on the existence of another application. That is, an application may not work if the application it depends on is not installed in the given system. Packages contain information regarding these dependencies. They also have attributes such as owner, version number, permissions, group ids, files etc.*

*Currently, Package management software tools provide a textual representation of the package information. This form of representation makes it rather tedious to visually decipher relationships between packages and realize patterns in these relationships. Since these images are important for system analyzers and distribution developers to analyze the system, a visualization of the package dependencies is desired.*

*This thesis report describes the construction of a visualization tool to help distribution developers, analyzers as well as curious operating system users realize the relationships between packages. This report attempts to achieve an acceptable visualization by researching graphs and applying these to visualize the dependencies between the packages.*

*Furthermore, this report identifies the applications and limitations of the visualization tool and examines another approach to address these limitations. This report also discusses an attempt to further improve the visualization by using certain improvement techniques.*

# DEFINITIONS

**Capability:** A capability is a functionality that the package claims it provides.

**Dependency:** The relationship between different packages.

**Edge:** A directed graph element representing the type of dependency.

**Group:** A collection of packages.

**Node:** A graph element representing a package/group/capability.

**Package:** Collection of related files along with a script containing file metadata.

**Provides:** Functionality that a package claims it provides.

**Requires:** Functionality that a package claims it requires from another package.

**RPM:** Red Hat Package Manager

**Sub-group:** A group within another group.

**Sub-package:** One of the several package files created from one package.

Chapter 1

# Introduction

Operating Systems consist of a large amount of complicated software. They offer a number of services to application programs and users. These application programs may be dependent on one another wherein, the proper working of one application may depend on the correct installation of another. In UNIX based operating systems, software is enveloped in *packages*. These packages are managed by Package Management Software (PMS) tools.

Various PMS tools exist, providing a certain amount of information about the packages. However, they do not always provide the user with a convenient overview of the system. Current package management software tools provide textual information that do not clearly show details about properties and dependencies between packages. Therefore, it would be beneficial to have a tool that can interactively visualize all the information conveniently, for users to be able to analyze the system.

This master's thesis report describes the construction of a tool that enables its user to have a graphical overview of the system's packages and their dependencies. This will enable the users to analyze the system, its properties, and help make informed decisions during the development of an application or operating system distribution.

## 1.1 Package

In UNIX based operating systems such as Linux, software applications are primarily distributed in the form of source code provided in zip-like archives called *packages* [2]. In order to install a software application, the source code has to be unpacked from its package, configured to support the options that the system wants and then compiled. The installation is completed by configuring the permissions of each of the files of the compiled package and finally putting these files in the required destinations of the file system. Software applications can also be distributed in the form of already compiled binary files. These can also be installed in a similar manner without performing the compilation.

Un-installation of software is carried out by removing the files and re-configuring the system options. For both installation as well as un-installation, the system is checked to verify

consistency. For example, dependencies between packages, version numbers, etc need to be verified such that the system complies with the requirements to install or un-install the software. Performing all these steps manually by hand is cumbersome as it involves hand typing different commands to perform different checks, installations, etc. In order to automate the process, a PMS tool is used.

Typically, a *package* contains:

1) A collection of related files, which together have a common use.
2) A script that provides all the metadata about those files necessary to install the software.

It is also possible for a package to have one or more sub-packages. A sub-package is essentially another package that is defined as one of the several package files. For example, a client server program may come in two flavors, a client part, and a server part. Therefore, different types of packages need to be created. In order to reduce redundant code, sub-packages are used. In this example, the main package contains the functionalities common to both client as well as server while the sub-packages include more specific functionalities.

## 1.2 Package Management Software

PMS ensures a proper software installation of a software application by checking the dependencies of the software and notifying the user if there are any conflicts or missing dependencies [2]. PMS also ensures the proper un-installation of the software application. Furthermore, if a new update for an already installed software application is available, the PMS ensures that the application is properly updated. PMS can also perform the necessary consistency checks while installing, updating or un-installing a software application. Some PMS tools have the ability to resolve consistency failures automatically, while others notify the user for manual resolution. Figure 1.1 is a schematic diagram describing the working of PMS for software installation.

In Linux, the PMS maintains a database of information regarding all the files of the system and applications installed in the system. When a new software application is installed, this database is updated with information regarding the new application and the location of the application's files. The database also provides the information necessary to perform consistency checks such as dependency or version conflict checks. Thus, the PMS has all the ingredients necessary to efficiently manage the packages of the system. There are a variety of PMS tools available in Linux. These include RPM, Synaptics, Adept, etc. In this report, we concentrate on RPM (Red Hat Package Manager).

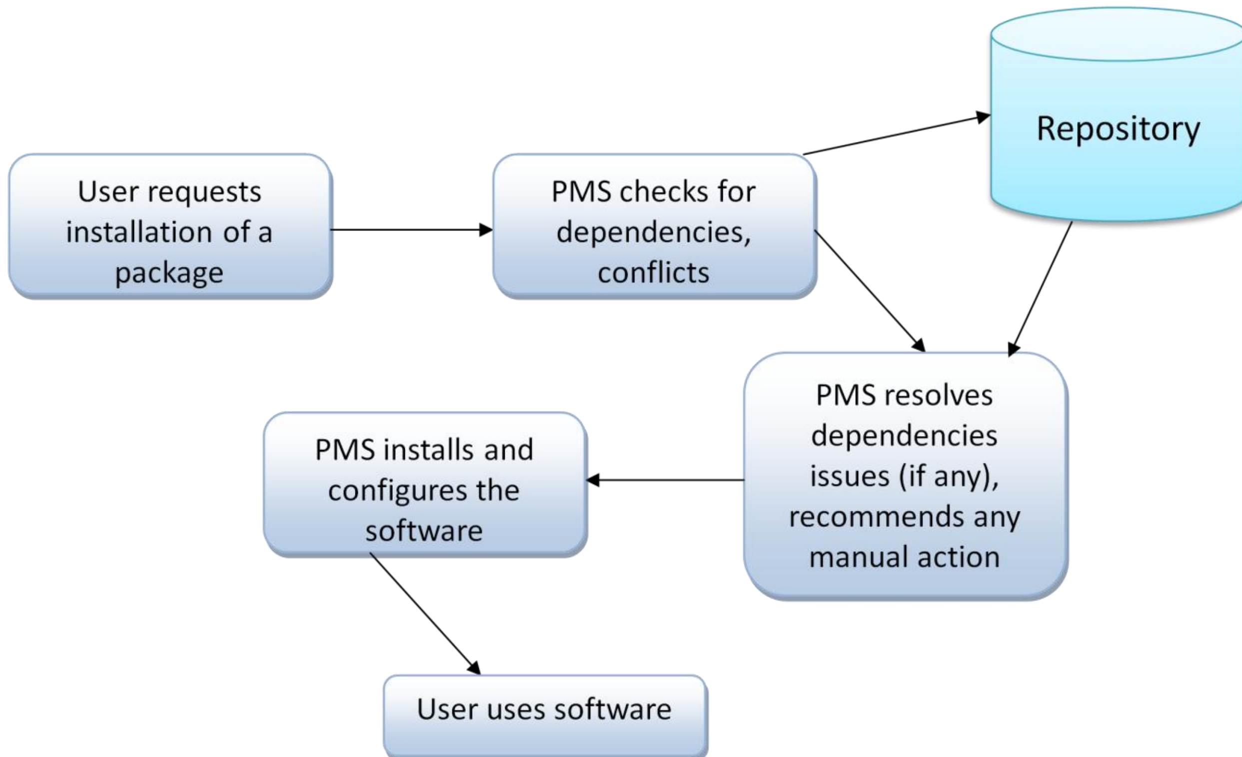



***Figure 1.1*** *Installation of a software using PMS.*

*The PMS installs a package by first determining whether the required dependencies are met using the information from the repository. If there are any missing dependencies or conflicts, the PMS notifies the user and recommends a solution. When all the dependencies and/or conflicts are resolved, the PMS installs the software and configures the system. The user than then use the installed software.*

## 1.3 User Questions

Besides performing the necessary verifications during the installations and modifications of any software application, PMS tools also provide details about the installed applications, such as their names, version numbers, and relationship with other packages. However, the presentation of this information is mostly textual and therefore it is not easy to visually analyze the system. In order to fulfill this requirement, we construct a tool that will visualize the necessary information.

To determine what the visualization tool is expected to achieve, it is important to know what information a target user would want to retrieve using such a tool. We assume our target users to be system analysts, application developers and distribution developers of the Linux operating system.

The following user questions are formulated:

1. What is the relationship between two user-chosen packages?

2. What is the impact on the system if a user wants to remove un-install or modify a package?
3. If one package is removed, is it possible that another package can now be removed without hindering the consistency of the system?
4. What are the statistical details of the package? That is, number of dependant packages, number of required packages.
5. Are there packages that do not depend on other package?

## 1.4 Requirements

The goal of this project is to implement a tool that enables the users of the system to have an overview of the entire system, the relationships between the different packages and the impact of any modification or conflict.

The following are a set of requirements that the tool is expected to fulfill:

1) The tool should be able to visualize the RPM package dataset by showing the required dependencies.
2) The user should be able to view specific dependencies between packages.
3) The user should be able to the entire package set and all their relationships.
4) The visualization is required to be reasonable fast, detailed, and interactive.
5) The visualization should be tidy, clutter-free and easily readable.

## 1.5 Report Outline

In this chapter, we introduced the concept of packages and tools that manage these packages. In chapter 2, we discuss a specific package management tool. We also discuss the data that is managed by this tool, how this data is retrieved and converted into a format that can be visualized. Chapter 3 discusses the visualization tool that we construct to visualize the package data. It discusses the tool features, its applications and limitations. In order to address some of the limitations of our tool, we discuss a solution in chapter 4. We also try to improve the visualization further by researching certain algorithms. The visualization results of our findings are discussed in chapter 5. Finally, we conclude the report by mentioning the future possibilities for improving the visualization.

Chapter 2

# RPM Package Dataset

In the last chapter, we discussed packages and the working of PMS tools. In this chapter, we look under the hood of one such tool – Red Hat Package Manager (RPM). We introduce this tool in Section 2.1 followed by a brief note on its design goals in Section 2.2. We provide a detailed description of the structure of the RPM dataset in Section 2.3. Finally, in Section 2.4, we describe how we retrieve this data and format it in a way such that it can be visualized graphically.

## 2.1 Red hat Package Manager (RPM)

There are various Package Management Software tools available for various Linux distributions. Red Hat Package Manager (RPM) is one such tool that comes along with most Red Hat based Linux distributions. The file format RPM is the baseline package format of the Linux standard base[2].

RPM consists of a single database that contains all the meta-information of the installed packages. The database keeps track of all the files that are changed or created when a user installs an RPM package. It also enables the user (via RPM) to reverse the changes and remove the package later.

RPM packages are made publicly available using an RPM Repository. Typically, such a repository contains thousands of applications. A limitation of RPM is that it does not follow dependency information automatically [2]. i.e., while installing a package, if a dependency requirement is not met, RPM simply notifies the problem to the user without automatically trying to handle the situation. However, there exists another tool called *yum* [14] which addresses this limitation but that is out of the scope of this report.

## 2.2 RPM Design Goals

There are various requirements [2] that the Red Hat team used in their design of RPM:

- **Easy installation and un-installation of packages:** One of the goals of RPM is to make it possible for anyone to install packages. The same holds true from removing packages from the system.

- **Easy to verify that the package is installed correctly:** RPM must ensure that the installation of the package is done correctly, by checking for missing dependencies and conflicts.
- **Easy for the Package Builder:** Another design goal of RPM is to make it easy for a package builder to build packages.
- **Make it start with the original source code:** Starting the package building process with the original source code makes it possible to separate the changes required to build the package from any changes implemented to fix bugs, add new features, or anything else. RPM allows the application of only original "package building" changes to the newer software, thereby making the task of updating packages easier.
- **Make it work on different computer architectures:** Since RPM makes it easy to start from a program's original source code, add the changes necessary to get it to build, and produce a package for different architectures in one step, this feature can be pretty useful.

## 2.3 Structure of an RPM Package

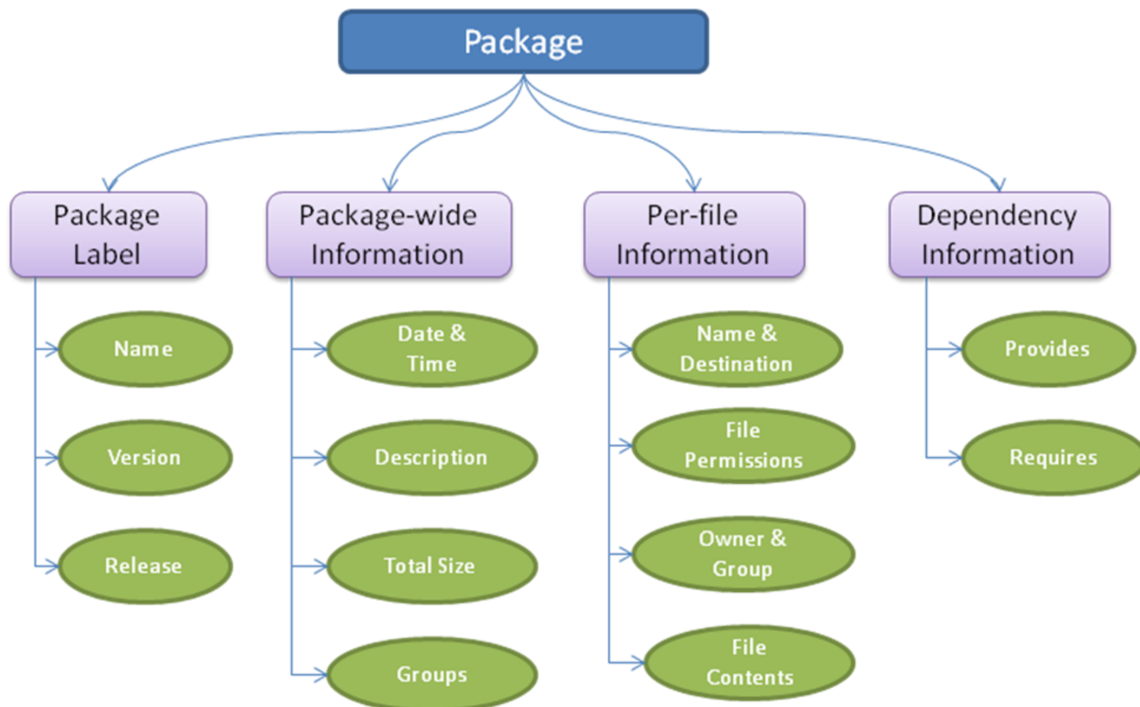



***Figure 2.1*** *Structure of an RPM package*

An RPM package contains a variety of information, which is structured as shown in Figure 2.1. The following subsections provide a brief description on the structure.

### 2.3.1 RPM Package Labels

Every RPM package has a specific set of information that uniquely identifies it. This information is called *package label*. For Example, *nls-1.0-1* and *perl-5.001m-4*. There exists a naming convention called RPM's package labeling convention for labeling packages. Every package label begins with the name of the software that the package contains. In the above example, *nls* and *perl* are the names of the software contained within the packages. The name could also be the name of the group of the related programs that are bundled together by the package builder.

The second component of the label is an identifier that describes the version of the software being packaged. In the given examples, *1.0* and *5.001m* describes the versions of *nls* and *perl* respectively.

The last component of the package label is the *package release*. This is the most unambiguous part of the package label. It reflects the number of times that package has been rebuilt using the same version software. Normally, rebuilds are made due to the bugs uncovered after the package has been used for a while.

### 2.3.2 Package-wide Information

Package-wide information is typically, general information that is contained within each package. This information includes terms such as:

- Date and time the package was built
- A textual description of the package contents.
- The total size of all the files that are installed by the package.
- Information that allows the package to be grouped with similar packages.

### 2.3.3 Per-file Information

Each package also contains information specified to every file that is contained within the package. This information includes:

- The name of every file and its destination on installation.
- Each file's permissions.
- Each file's owner and group specifications
- Contents of each file.

### 2.3.4 Dependency Information

In Linux, packages may depend on other packages. This is advantageous as it reduces redundant code between different packages. It is also benefits the process of updating packages. For example, a security fix in one of the main packages can update all the applications that make use of the updated package. Packages advertise their information and these can be accessed by using RPM dependency commands.

Every package claims to be able to perform one or more functionalities. We use the term *capability* to describe the functionality of packages. In most cases, a capability is either a file or another package. It could also be a text string. When a package is built by RPM, if any file in the package's file list is a shared library, the *soname* of the library is automatically added to the list of capabilities the package provides.

There are two types of dependency information

- **Provides:** This is a list of all the capabilities that a package claims it provides.
- **Requires:** This is a list of all the capabilities that are required by the package. RPM gets this information by running **ldd** on every executable program in the package's file list. **ldd** is a Linux command that provides a list of the shared libraries each program requires.

RPM performs various kinds of consistency checks using the dependency information provided by the RPM database. Version dependency check, for example, is one where a package may depend on the capability of a specific version of another package. Conflict check is performed to check for Packages that provide capabilities that may interfere with those of other packages. Furthermore, RPM can also check if a package provides a capability that is an older version of a capability offered by another already installed package. Figure 2.2 shows an example of package dependency.

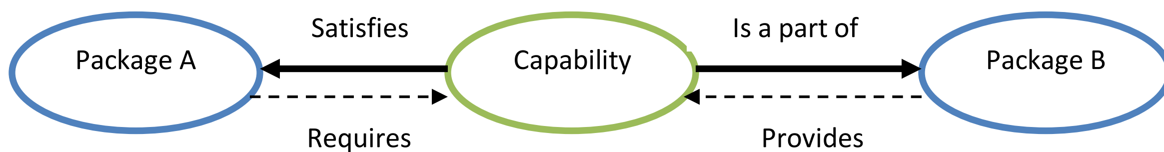



***Figure 2.2*** *Package A requires a capability that Package B provides*

*The capability that satisfies the requirements of Package A is a part of Package B.*

## 2.4 Retrieving the RPM Package Data

In order to visualize the RPM packages and their properties, we first need to retrieve the package information from the RPM repository and structure it accordingly. Figure 2.3 shows the process of retrieving the package information and structuring.

***Figure 2.3*** *XML to GRAPHML conversion*

We use the Createrepo program described in Section 2.4.1 to retrieve all the package information. The retrieved information is structured in the form of an *eXtensible Markup Language* (XML) specification file. In order to use this information in our visualization tool, we need to convert the XML specification into a GRAPHML Specification [13]. We implement a restructuring tool written in JAVA to do this. The restructuring tool takes the XML file as the input, and reformats the information into a GRAPHML file. We do this process in order to abstract the data to a general graph specification. This specification is required in order to draw the visualization.

### 2.4.1 Createrepo

*Createrepo* is a program that creates an XML-based RPM metadata repository from a set of RPM packages. We use this program to obtain an initial structure of the RPM packages. The linux distribution we use to retrieve the RPM packages is *Fedora 10*. *Createrepo* retrieves all the packages that are installed in the current system.

Example: createrepo -u /path/to/rpms

The path above is the base URL location of all the files. (Not used by any clients at this time). The command outputs an XML file containing the information of all the RPM packages in that path [5]. Figure 2.4 shows the XML skeleton of the package information structure. Information about the package is structured within XML tags. For example, *name <name>, architecture <arch>* tags etc.

### 2.4.2 Graph Structure

Once the dataset information is retrieved using the *createrepo* program, we need to re-format it in a way such that it can be visualized by our visualization tool.

The visualization of RPM package information centers on the mathematical notion of a directed graph. A directed graph is defined as a graph consisting of two sets, a node set and an edge set. Nodes represent any kind of entity while edges represent the relationship between the nodes [8]. In RPM terminology, nodes are packages, sub-packages, groups, subgroups or capabilities. Edges, on the other hand, are of two types, namely, *inclusive* and *dependency*. Inclusive relationship is when a node is contained within another node. A dependency relationship is one in which a node depends on another node. That is, one package requires a capability from another package or provides a capability to another package. Both

provides and requires relationships are forms of dependency relationships. Figure 2.5 is an example graph containing a set of nodes and edges.

In order to visualize the dataset in the form of a directed graph, the XML specification retrieved from *createrepo* needs to be converted into a graph specification to serve as an input to our visualization tool. Therefore, we use a restructuring tool to convert the XML specification into GRAPHML specification. Figure 2.6 shows the GRAPHML specification for RPM package information.

The GRAPHML specification structures the RPM package properties in terms of nodes and edges. Every package, sub-package, group and sub-group are identified within *<node>* tags and every relationship, both inclusive and dependency, are identified within *<edge>* tags. In the next chapter, we use this specification as an input to create the visualization of packages and their properties.

```
<?xml version="1.0" encoding="UTF-8" ?>
  <metadata packages="" xsi:schemaLocation="http://linux.duke.edu/metadata/common
  hmm.xsd" xmlns="http://linux.duke.edu/metadata/common"
  xmlns:rpm="http://linux.duke.edu/metadata/rpm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <package type="">
              <name />
              <arch />
              <version rel="" ver="" epoch="" />
              <checksum type="" pkgid="" />
              <summary />
              <description />
              <packager />
              <url />
              <time build="" file="" />
              <size archive="" package="" installed="" />
              <location href="" />
              <format>
                    <rpm:license />
                    <rpm:vendor />
                    <rpm:group />
                    <rpm:buildhost />
                    <rpm:sourcerpm />
                    <rpm:header-range start="" end="" />
                    <rpm:provides>
                          <rpm:entry name="" />
                    </rpm:provides>
                    <rpm:requires />
              </format>
        </package>
```

***Figure 2.4*** *XML Package information Structure.*

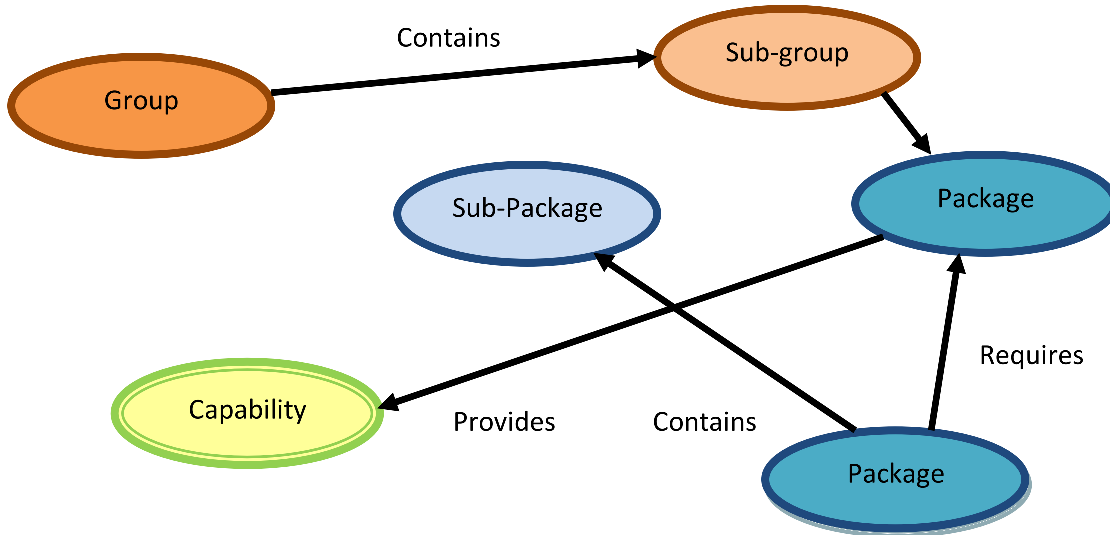



***Figure 2.5*** *An example directed graph for RPM information.*

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Sample XML file generated by XMLSpy v2008 rel. 2 (http://www.altova.com)-->
<!DOCTYPE graphml SYSTEM "dtd2.dtd">
<graphml>
	<graph>
		<key id="" attr.name=""/>
		<node id="">
			<data key=""/>
		</node>
		<edge source="" target=""/>
	</graph>
</graphml>
```

***Figure 2.6*** *GRAPHML Package Information Structure*

Chapter 3

# The Visualization Tool

The previous chapter discussed the retrieval and structuring of the RPM package information into a GRAPHML specification. This chapter discusses the construction of the visualization tool which will use this specification as an input to visualize the package information in the form of a graph. Section 3.2 discusses the force directed method of visualizing a graph. Section 3.3 introduces the notion of filtering nodes and edges depending upon user needs. The visualization tool is realized using a graph visualization toolkit called PREFUSE [1], which is briefly discussed in Section 3.6. The features of the tool are specified in Section 3.5. Finally, the applications and limitations of the tool are discussed in Section 3.7 and 3.8, respectively.

## 3.1 Abstract Data

The process of visualization starts with the abstract data that is to be visualized. The GRAPHML specification structures the properties and relations of packages and groups in the form of nodes and edges respectively. These entities form the abstract data that the visualization tool uses to visualize the data in the form of a graph.

## 3.2 Force Directed Layout

After the abstract graph entities are identified via the GRAPHML specification, we can then draw the graph using a suitable layout. There exist various graph drawing layouts, from which, we use the force directed layout because of its simplicity, interactivity, popularity and the quality of results. Graphs are depicted with vertices as points and edges as straight or curved line segments connecting the points. In our visualization tool, we concentrate on directed graphs with straight directed edges. Force directed layouts draw graphs in a way that fulfills the criteria [4] given below:

- Distribute the vertices evenly in the frame
- Make the edge lengths uniform.
- Reflect inherent symmetry.

- Conform to the frame. That is, make sure the visualized nodes do not move outside the visualization area.

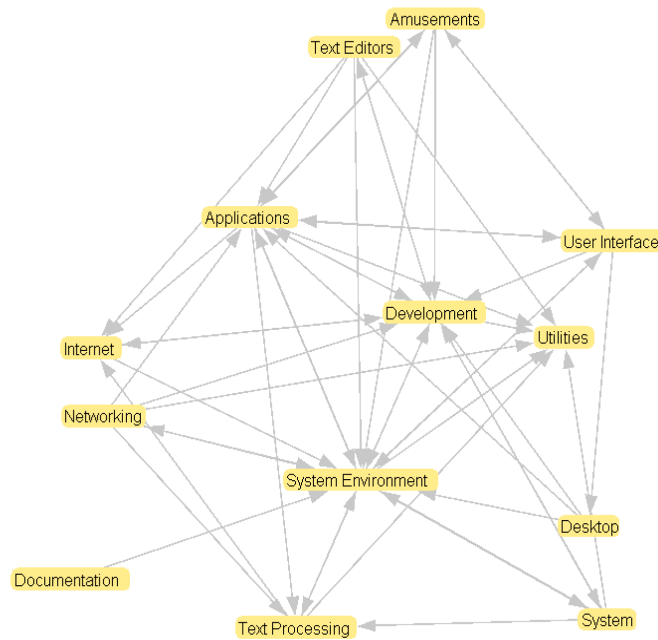



***Figure 3.1*** *Graph drawn using force directed layout*

*The nodes are the groups of RPM packages. The edges show the dependency direction. Outgoing edges indicate that the group depends on other groups.*

In force directed layouts, the simulation of a graph is created such that the nodes exert anti-gravity to push each other away. This anti-gravity force helps to avoid the overlapping of the visualized nodes. The edges behave like springs to position the nodes such that they are closer to their neighbors. This helps in maintaining a sense of symmetry in the visualization. Friction and drag forces are used to stabilize the visualization. Figure 3.1 shows an example of a graph, drawn using force directed layouts.

## 3.3 Filtering

The graph to be drawn can be filtered such that only a necessary subset of the nodes and edges are visible. Formally, *filtering* is the process of mapping the abstract data into some canonical form, which is then visualized [1]. The visual items (nodes and edges) are generated from the abstract data and are visualized in the display area. Different filters are used for different conditions and for different visual items based on the requirements of the user. For example, filters can be created so that the visualization displays only nodes that have a particular number of incoming edges.

## 3.4 User Interaction

A requirement of the visualization tool is interactivity. By enabling user interactivity, nodes can be repositioned, removed, or highlighted. The forces of the force directed layout can be adjusted interactively such that edges and the nodes are visualized clearly. The visualization also responds to mouse actions, which can be used for panning, zooming, and node selection.

## 3.5 Tool Features

The characteristic features of the visualization tool are described below:

### 3.5.1 Node Hierarchy

There exist a large number of RPM packages in a Linux system. Therefore, to conveniently visualize it, we use a *node hierarchy*. By using hierarchies, we enable the visualizations to provide more detail, and with less visual clutter while describing relationships between the packages, groups and capabilities. Figure 3.3 describes the hierarchy of the graph that we use in our visualization tool.

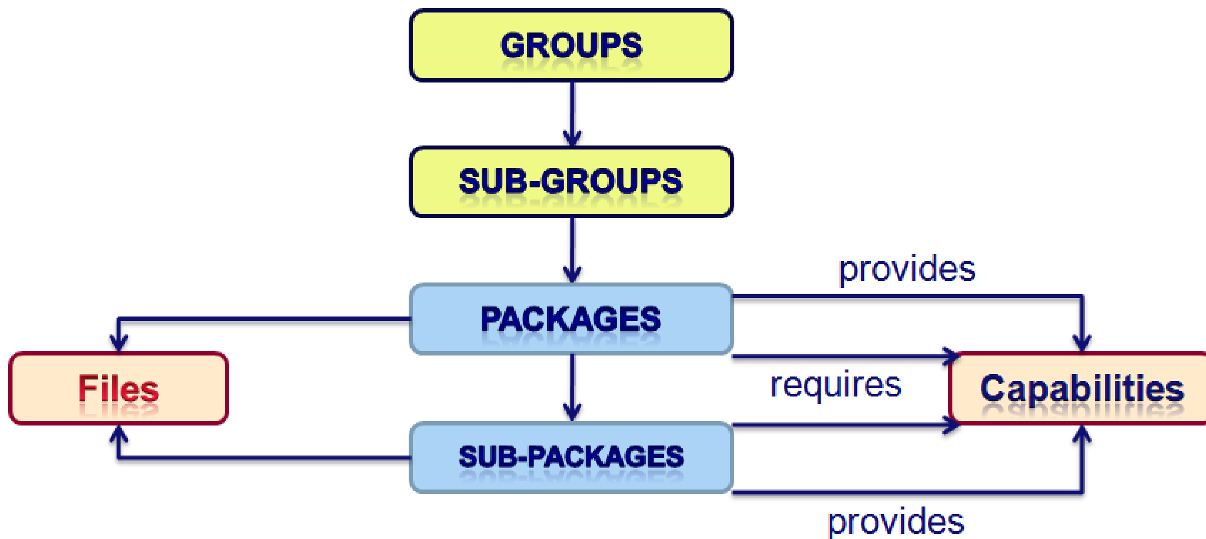



***Figure 3.2*** *Node Hierarchy.*

In the RPM package dataset, groups form the highest level of the hierarchy. Each group may have one or more sub-groups. Sub-groups in turn, contain packages. Each package contains a number of files. Based on these files, the packages claim to provide or require certain capabilities.

### 3.5.2 Views

Our Visualization Tool visualizes the RPM packages and their relations in three views:

- Group based view
- Package based view
- Capability based view

#### 3.5.2.1 Group View

In a Linux system, packages are grouped based on their area of application. Figure 3.1 shows the graph with groups as the nodes and edges as the relationships between them. The relationships between the groups are essentially the relationships between the packages within these groups. i.e. if *PackageA* from *GroupA* requires *PackageB* from *GroupB*, then *GroupA* requires *GroupB*. The direction of an edge signifies the direction of the dependency. An arrow head from *GroupA* to *GroupB* indicates that *GroupA* depends on, or requires *GroupB*.

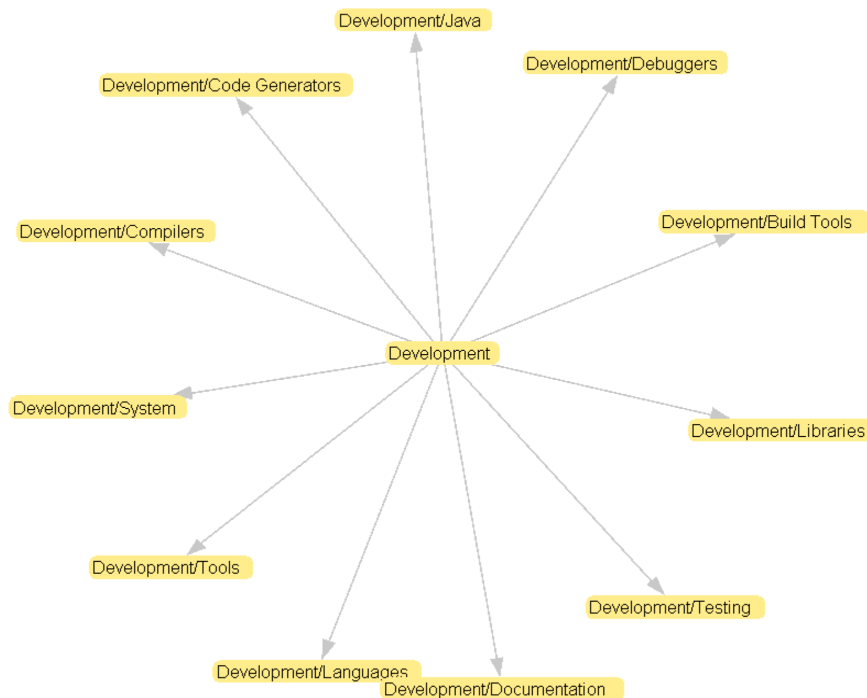



***Figure 3.3*** *Sub-Groups of the Development group*

*. The edges indicate the inclusion relationship between the Development group and its subgroups.*

In Figure 3.1, an edge from the *Application* group to *Internet* group means that there is at least one package within the *Application* group that depends on at least one package from the *Internet* group. Most groups also have sub-groups to add to the hierarchy. Figure 3.3 shows the sub-groups for the *Development* group. In this case, the edges are inclusive in nature. That is, the direction of the edge signifies the inclusion of one package within another. For Example, *Java* is a subgroup of the *Development* group. Therefore, there is an edge from *Development* to *Development/Java*.

#### 3.5.2.2 Package View

Package view is similar to that of the sub-group view shown above. This view shows the inclusion of packages within the subgroups. Figure 3.4 shows this arrangement. Here, the packages that belong to the group *Development/Debuggers* are shown. Using filters, we color the currently less visually important nodes grey. The edges here again indicate inclusion. For Example, *ltrace* is a package that belongs to the sub-group *Development/Debuggers*.

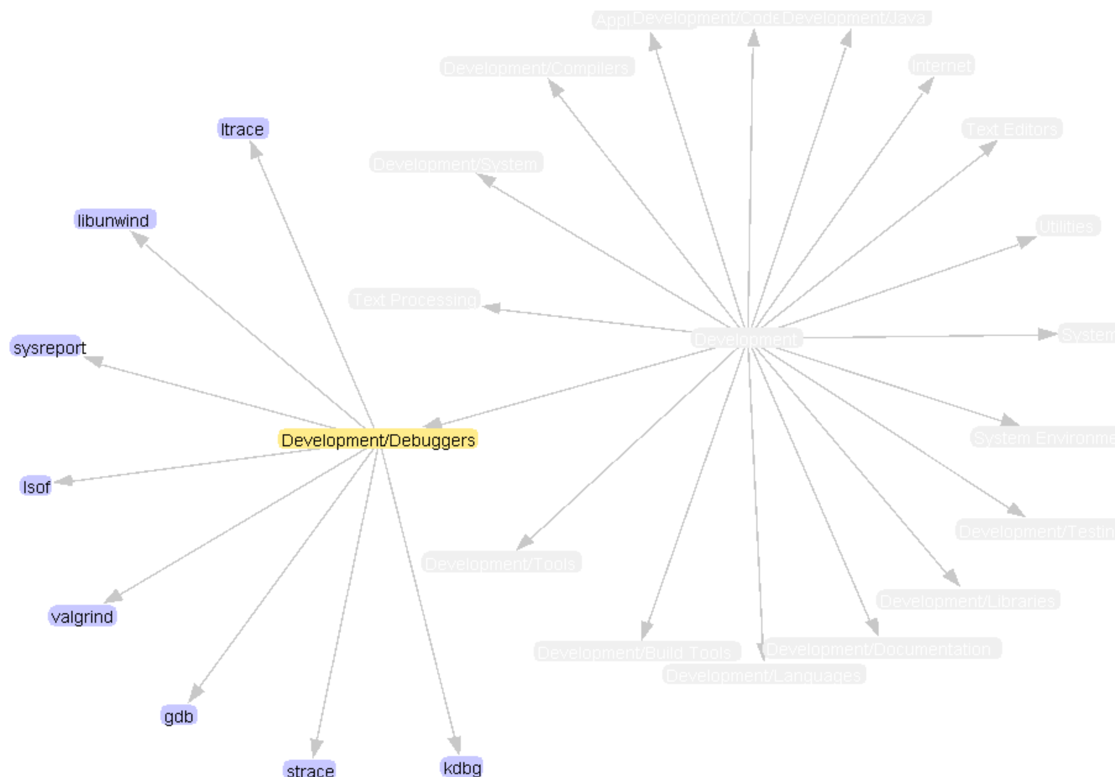



***Figure 3.4*** *Member packages within the Development/Debuggers subgroup.*

#### 3.5.3.3 Capability View

Every package has a set of functionalities it claims to provide. These functionalities are called capabilities. Figure 3.5 shows the capabilities of the *BackupPC* package. The edges indicate that the capabilities are provided by *BackupPC*.

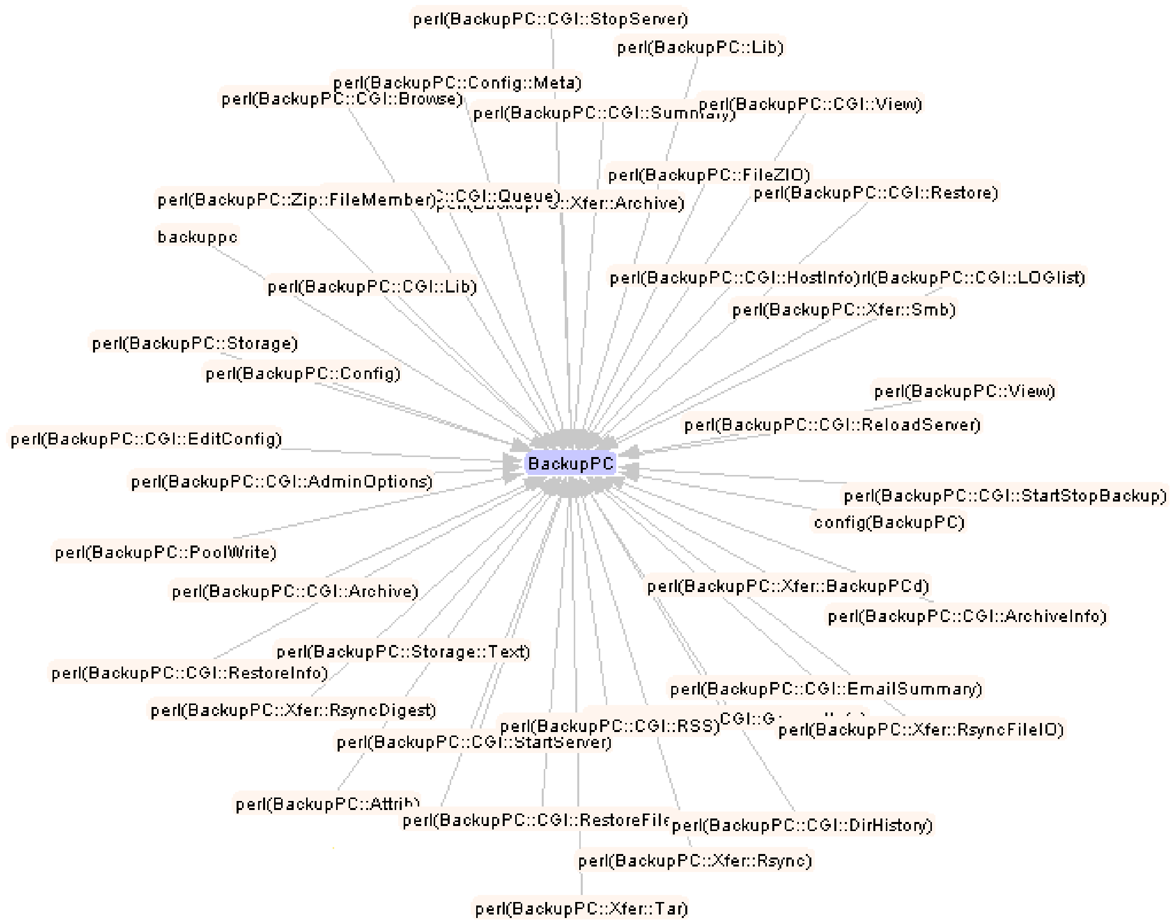



***Figure 3.5*** *Package capabilities*

*These are the capabilities that the BackupPC package claims to provide.*

### 3.5.3 Functionalities

The following subsections describe some of the functionalities of our visualization tool.

#### 3.5.3.1 Package Relationships

The visualization tool allows users to see the relationships between packages. Figure 3.6 shows the relationship between the package *bug-buddy* and all the packages it requires up to one level. It is also possible to increase the levels in order to see greater depth in the relationships between packages.

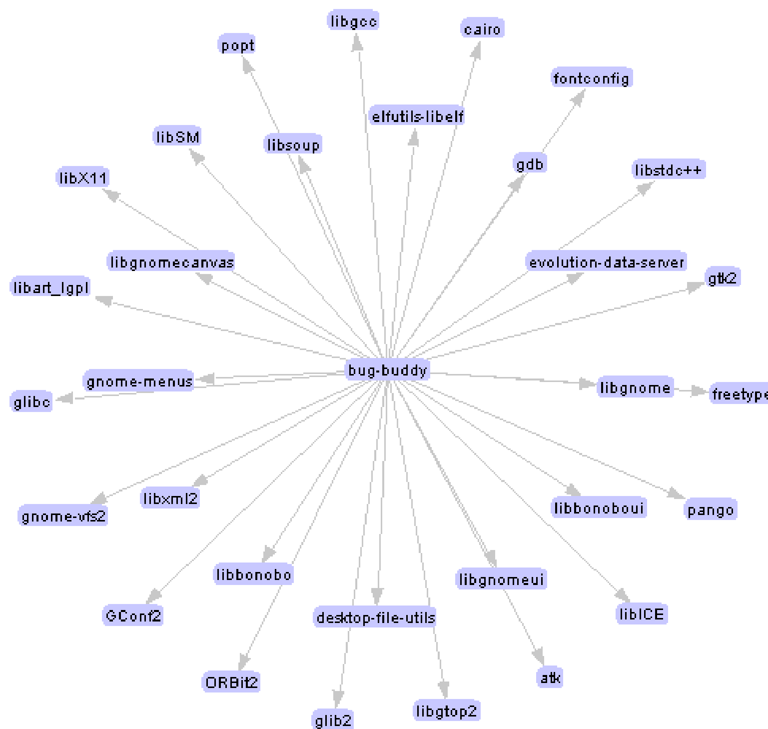



***Figure 3.6** Package relationships*

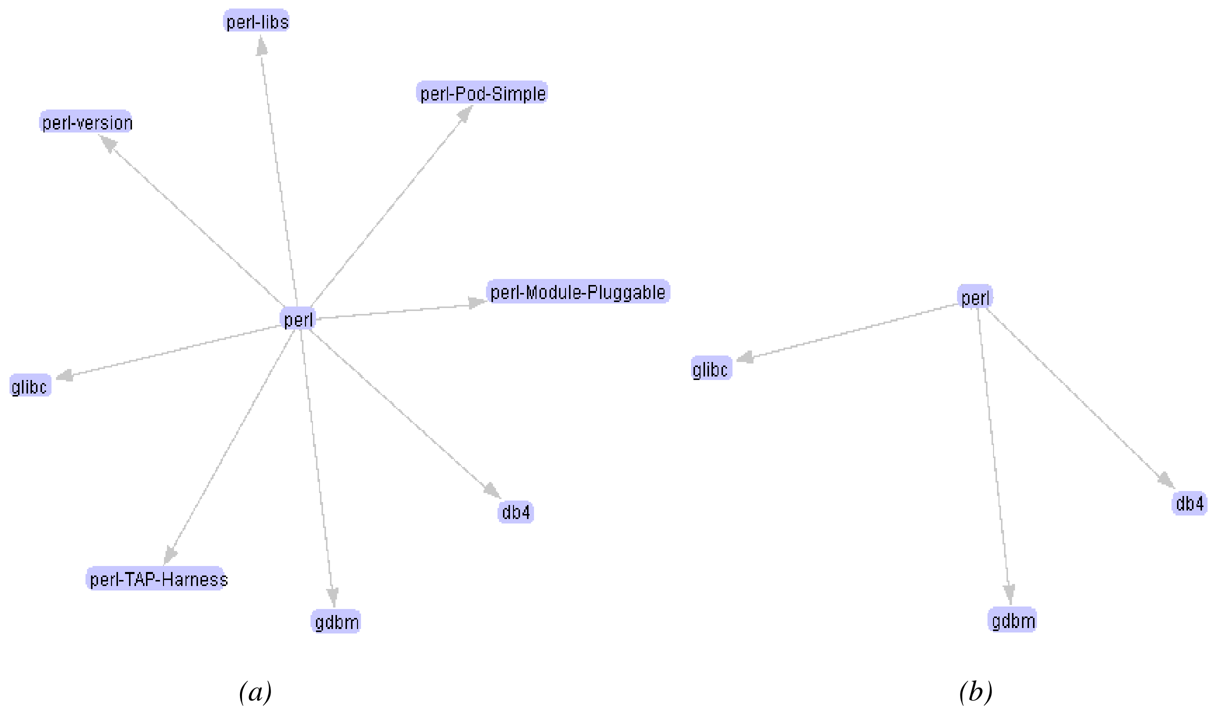
*. All packages that are required by bug-buddy are shown.*

#### 3.5.3.2 Package Hiding

Sub-packages are RPM packages that are part of other RPM packages. That is, they are built using another package's source RPM. The visualization tool allows users to hide/show these sub-packages in order to reduce visible clutter. Figure 3.7(a) shows the *perl* package along with its sub-packages. Figure 3.7b shows the *perl* package after hiding its sub-packages.

Some packages are required by almost every package in the system. Visualizing these packages along with their relationships causes a high visual clutter. For example, *glibc* is a package that is required by most packages of the system, either directly or indirectly. Figure 3.8 shows the clutter when *glibc* is not hidden.

On hiding *glibc*, all the edges that are connected to or from *glibc* are hidden. Also, the nodes that become disconnected from the graph after hiding *glibc* are also hidden. Figure 3.9 shows the graph when *glibc* is hidden. The visualization becomes slightly cleaner with lesser clutter.




*(a)* *(b)*

***Figure 3.7*** *Sub-package Hiding*

*. (a) All packages required by perl are shown. (b) Packages required by perl after hiding the subpackages of perl*

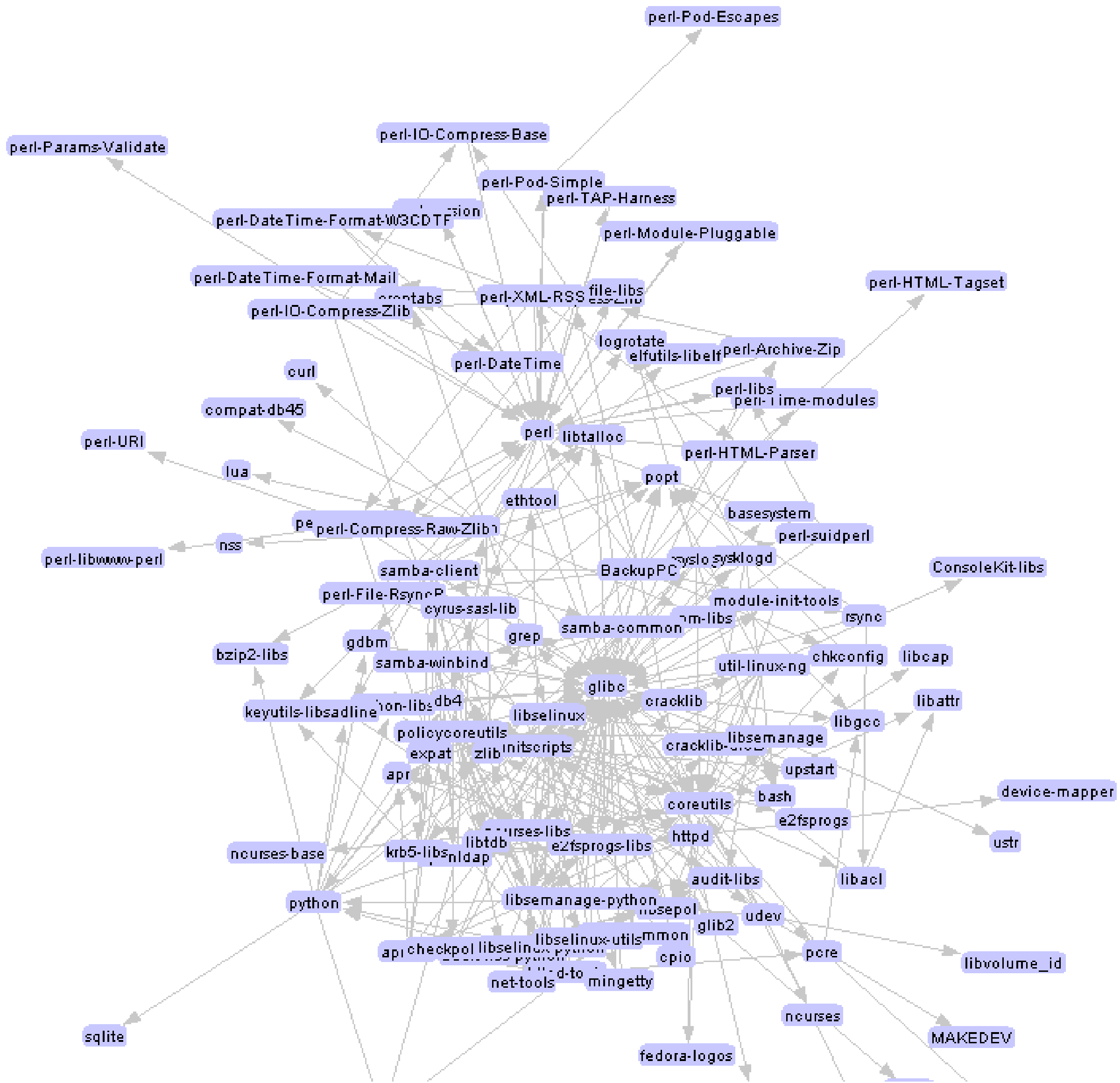



***Figure 3.8*** *Package Hiding*

*. The force directed visualization when glibc is not hidden. The visualization shows all dependencies upto three levels deep.*

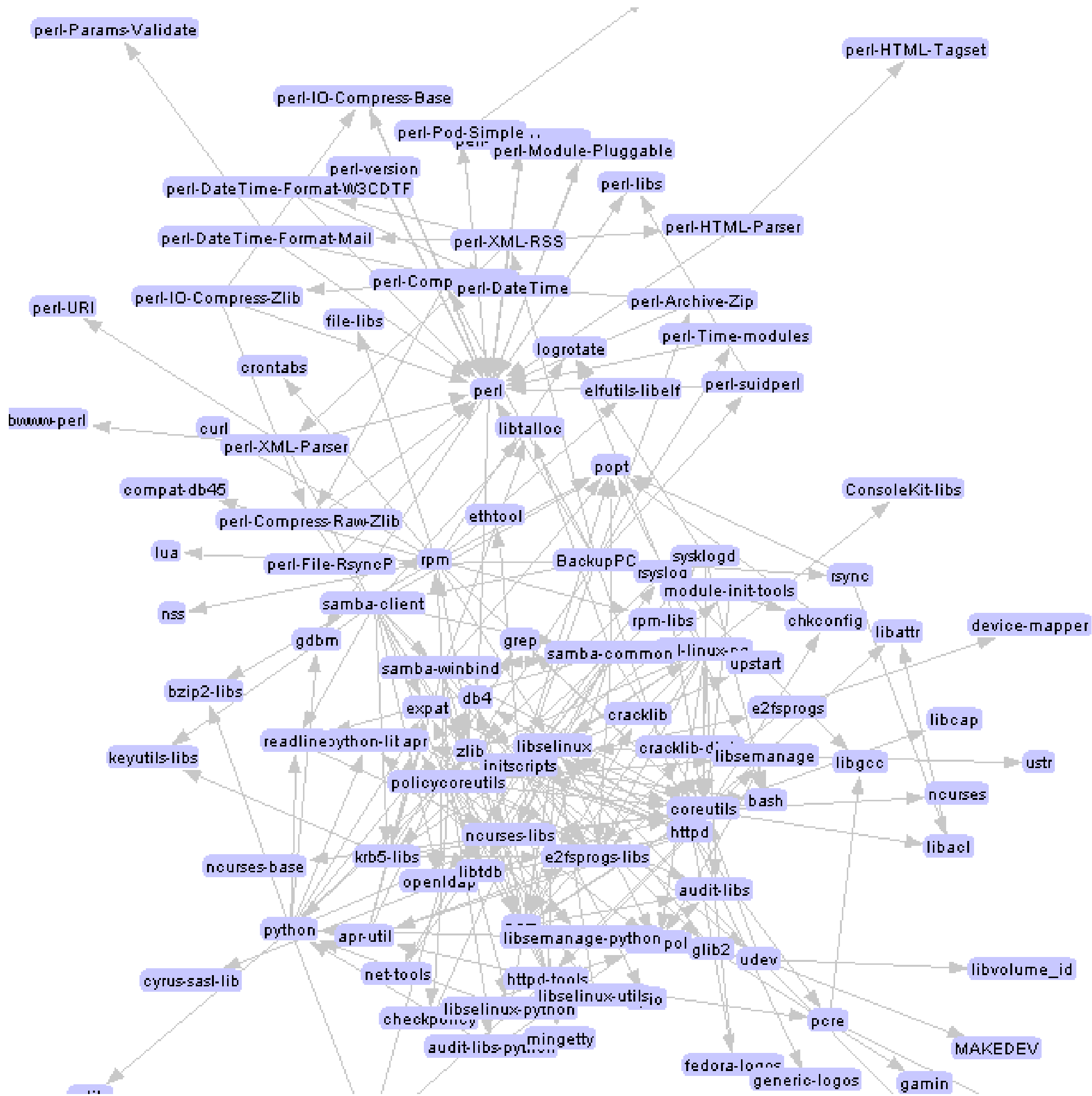



*Figure 3.9 The force directed visualization when glibc is hidden*

*. The visualization shows all dependencies upto three levels deep.*

### 3.5.3.3 Package Removal

For distribution developers, a desirable functionality of the visualization tool is to be able to see the impact of certain system changing operations. One such operation is *package removal*. The user would want to see the impact of attempting to remove a package from the system. The visualization tool shows whether a package can be removed without causing the system to become inconsistent.

Figure 3.10 shows the impact of removing the package *BackupPC* from the system. The tool checks to see if any other package requires its capabilities. If there are no dependant packages, then the removal of this package will not cause system inconsistency. Therefore, the tool colors the node green which means that the package can be removed.

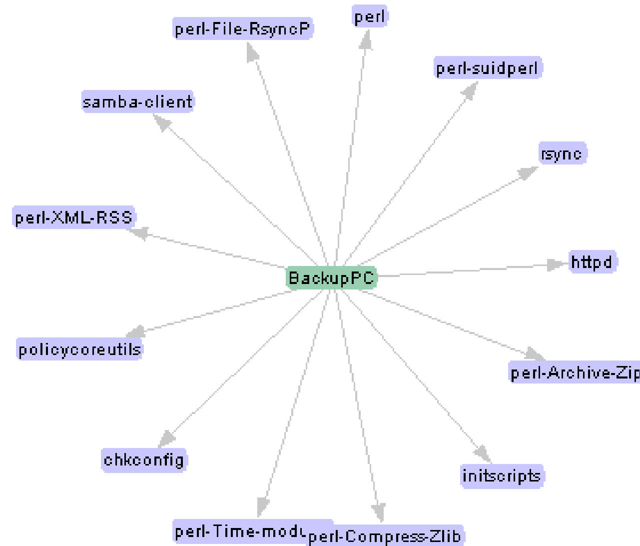



***Figure 3.10*** *BackupPC package removal*

The visualization tool also highlights the packages that can be removed after the current package is removed. Figure 3.11 shows the package removal path. Here, all the packages that can also be removed after *BackupPC* is removed are highlighted.

The above examples were for the removal of packages that were not required by any other package. An interesting question would be, "*What is the impact of removing a required package?"*. The visualization highlights the node and also shows the nodes that become inconsistent if the package is removed. Figure 3.12 shows this scenario.

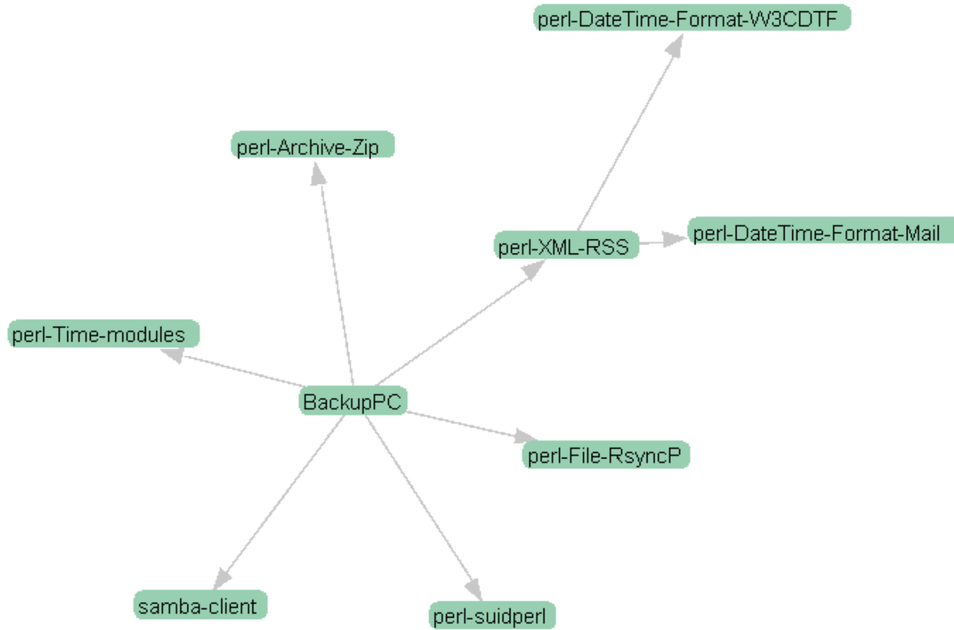



***Figure 3.11*** *BackupPC package removal path*

*. After BackupPC is deleted, its neighbors shown in the figure are also highlighted because they are now, not required by any other packages.*

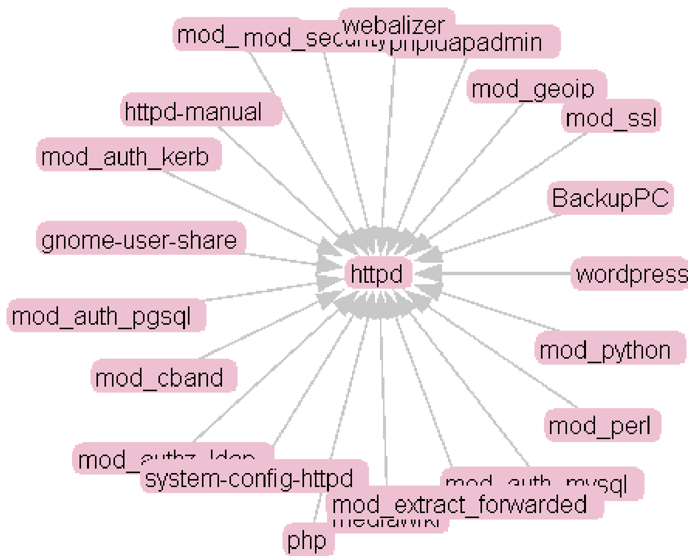



***Figure 3.12*** *Impact of removing the httpd package*

*. The visualization indicates that if httpd is deleted, the packages that require it (incoming neighbors) become inconsistent.*

#### 3.5.3.4 Relationship between two user chosen packages

The visualization tool can show detailed information about the relationship between two packages. Figure 3.13 shows the dependency between *BackupPC* and *perl* packages. The visualization also shows exactly which capabilities of the *perl* package *BackupPC* requires.

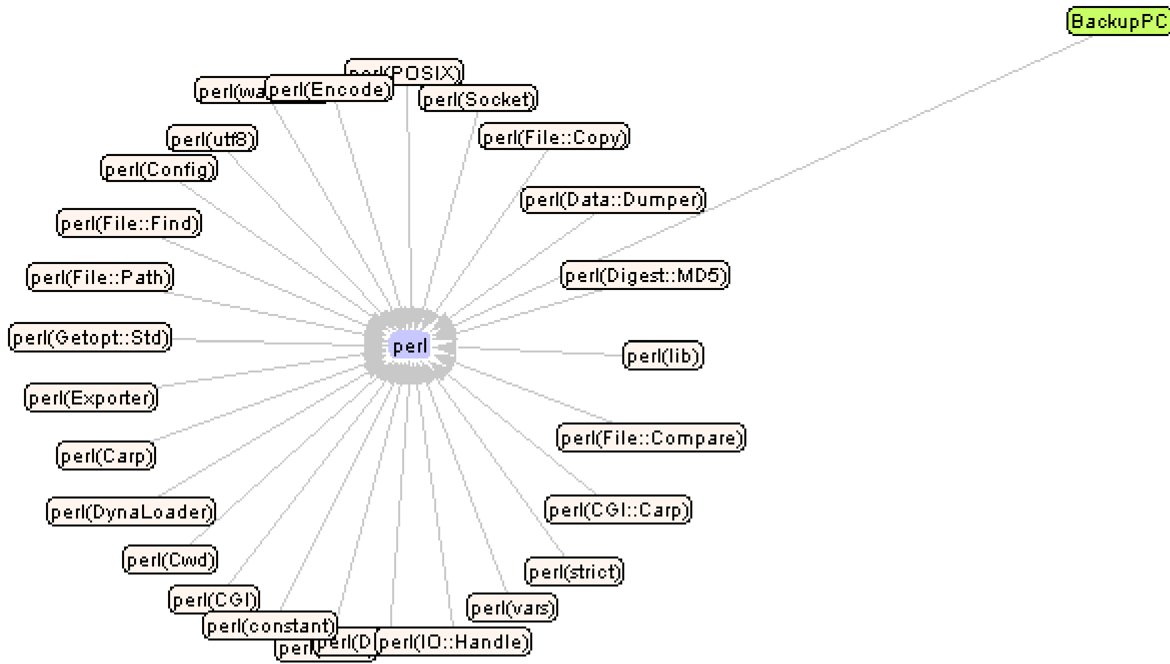



***Figure 3.13*** *BackupPC - perl package relationship*

Packages may depend on other packages either directly or indirectly. The visualization tool allows users to see such relationships between packages. Figure 3.14 shows an example of the relationship between *BackupPC* and *glibc*. Although *BackupPC* does not directly depend on *glibc*, there exists an indirect dependency between them through the *perl* package.

***Figure 3.14*** *Impact of removing the httpd package*

# 3.6 PREFUSE Visualization Toolkit

In order to draw the graph using the force directed layouts, we use an existing visualization toolkit called PREFUSE[1]. PREFUSE is an extensible user interface toolkit for visualizing both structured and unstructured data. It enables the realization of the abstract data into a graph by making use of the force directed layout as well as filtering. It applies scalable abstractions for mapping abstract data into visual forms and manipulating visual data in aggregate, supporting design in a modular and principled fashion. The toolkit's fundamental data structure is a graph – A set of entities and relations between them. PREFUSE also allows abstractions for filtering the input data into visualized content and using actions to process this content, allowing developers to manipulate the content in aggregate. It includes a rich library of layout algorithms, navigation and interaction techniques, integrated search, and more.

The primary goal of PREFUSE is to facilitate visualization design by reducing the implementation costs, allowing techniques to be combined in new ways, and promoting application modularity and extensibility. Figure 3.15 shows the PREFUSE visualization framework.

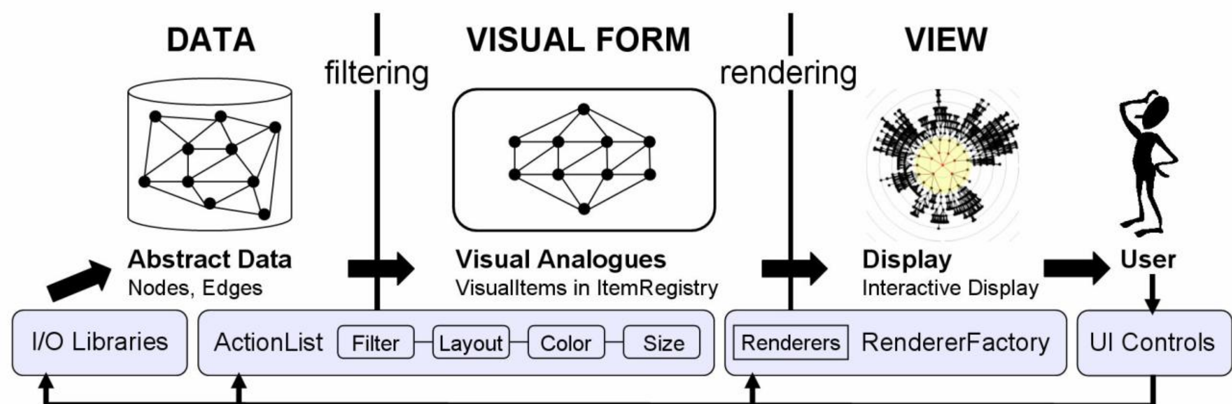



***Figure 3.15*** *[1] The Prefuse Visualization Framework*

The list of actions in the *ActionList* filters the abstract data into visualizable content and assigns visual properties such as position, colors, size, fonts, etc. to them. Renderer modules, provided on a per-item basis by a *RendererFactory* draw the *VisualItems* (nodes and edges) to construct interactive *Displays*. User Interaction can then trigger changes at any point in the framework.

# 3.7 Applications of the Visualization Tool

The visualization tool finds its use in the following applications:

- In order to analyze the contents of the operating system, system analysts need to be able to see specific and detailed relationships between packages. This tool provides a graphical view of the relationships and helps users see the exact dependencies between packages.
- The tool also enables users to see the relationships between nodes of the higher levels in the node hierarchy.
- The tool allows users to place any node as the center of attention thereby enabling the visualization of all the information pertaining to that particular node. Therefore, the tool visually simplifies the relationship analysis of specific nodes
- The tool enables the users to view certain system changing scenarios, such as removal of packages. The tool visualizes the impact of these operations on the system, helping users such as distribution developers make informed decisions about avoiding or including packages in their distribution.
- The tool allows the user to have full control over the visualization. The user can filter packages based on his or her needs. The user can also zoom, pan, or modify the visualization properties as desired.

# 3.8 Limitations of the Visualization Tool

Although the visualization tool provides satisfactory results in visualizing specific package relationships, it has important limitations:

- High running time. The force directed algorithms have a running time is $O(E)$, where $E$ is the number of edges in the graph. This is because the spring forces have to be calculated for each edge. Therefore, as the number of edges increases, the speed of the visualization decreases, thus losing interactivity.
- As the number of relationships between nodes increases, the visualization becomes cluttered and difficult to read.
- The tool fails to provide a complete overview of the system along with all its relations. In order to see patterns of relationships and certain trends in the system, a complete relationship view is required. Since force directed layouts are incapable of handling vast amount of data interactively, the tool fails to provide the visualization for such analysis.
- The visualization provides hierarchical views but cannot structure the visual nodes in a hierarchical fashion. The visualization provides a simple graph view of nodes and edges and hence, does not provide a definite structure to realize hierarchies.

# 3.9 Possible Solution

The visualization tool we constructed had the capabilities to visualize small number of relationships and specific package information. However, it failed to provide an acceptable visualization for the system and all its relationships as a complete entity. In order to analyze the complete dataset, we desire a

visualization that will show us the entire package and all its relationships in one view. We also desire a visualization that can structure the node hierarchy in a better way. In the following chapters, we discuss a possible solution that may be able to cater to these needs.

Chapter 4

# Circular Layouts

In the previous chapter, we visualized the RPM packages and their relationships as a graph, using a force directed layout. This method was advantageous when visualizing specific relationships between small sets of packages. However, when the number of visualized packages increased, the tool became slower, and visually cluttered. Therefore, there was a need to improve the visualization for larger sets of packages. Moreover, there was a need to clearly show the hierarchical structure of the dataset and the relationships between higher level members. In this chapter, we consider a possible solution to the problem – *circular layouts*. We use circular layouts because they provide a compact representation of the data. The nodes are placed on a circle, making the layout highly regularized, and achieving a clear depiction of each node. Furthermore, circular layouts can visualize large number of nodes and relationships in a much more efficient manner as compared to force directed layouts. These properties make circular layouts a superior choice as compared to force directed layouts, for visualizing larger datasets.

We introduce the concept of circular layouts in Section 4.1. Since our RPM dataset is hierarchical in nature, we need to structure the circular layouts to include hierarchy. This is discussed in Section 4.2. We also discuss two visualization improvements to the hierarchical layout: one is node ordering, introduced in Section 4.1 and the other is Edge Bundling, introduced in Section 4.2.

## 4.1 Circular Layouts

Circular layouts are one of the most important and oldest conventions used for drawing graphs [3]. In these layouts, nodes are drawn on the circumference of a circle, while the edges are line segments passing within the circle. Figure 4.1 shows a graph with a circular layout. Here, the nodes are placed at equal intervals on the circle and their relationships are indicated by the edges connecting them, within the circle.

Since our dataset is a multilevel hierarchy consisting of parent child (inclusion) relationships as well as non hierarchical adjacency (dependency) relationships, a single level circular layout is not sufficient as it does not indicate hierarchical information. There exist some tree visualization methods that help achieve a hierarchical context for circular layouts. Balloon tree [3] is an example

of one such method. Here, the sub-trees of a node lie within circles, and these circles are themselves placed on the circumference of a circle. Figure 4.2a shows an example of a balloon tree. The radii of the circles are defined by the number of descendants they have. Therefore, nodes that have a larger number of descendants will get a larger share of the display space [3]. Although this technique utilizes space efficiently for visualizing hierarchical data, it does not provide an effective visualization for adjacency relationships. Figure 4.2b shows this scenario. The visual clutter is noticeable when adjacency edges are visualized over the balloon tree.

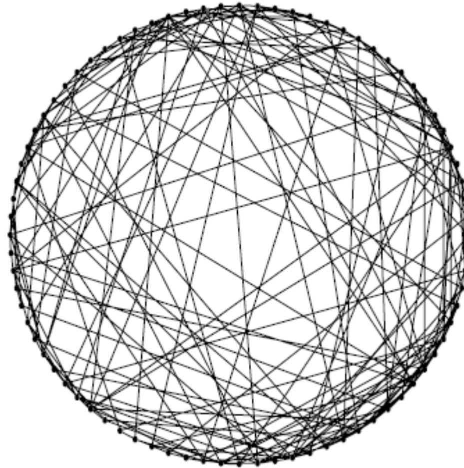
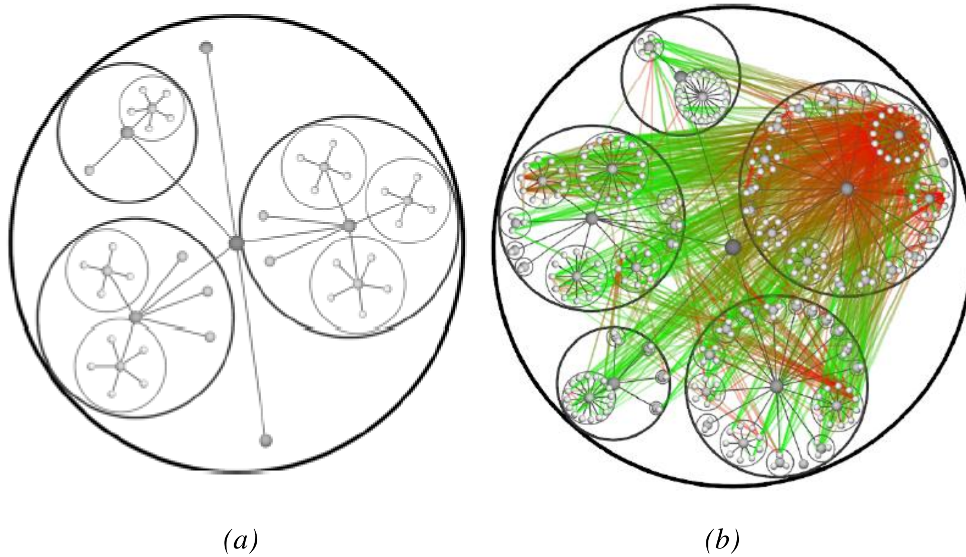


***Figure 4.1*** *Circular layout of a random graph [9]*

There is a tool that allows the hierarchical representation of data on a circular layout – ExtraVis. ExtraVis [11] is a beta-stage, OpenGL-based software visualization tool that enables the visualization of gathered data from a software system in a condensed manner, while still being highly scalable and interactive. We concentrate on the circular view feature of this tool to visualize the relationships between groups and packages and dependencies between packages. Figure 4.3 shows an example visualization of a hierarchical circular layout.

## 4.2 Improving Circular Layouts

In a circular layout, the nodes of a graph are constrained to distinct positions along the circumference of a circle, with edges connecting them passing within the circle [9]. This means that edges may cross each other in such layouts. A reasonable number of crossings may be acceptable but if the number of edges is large, the number of crosses may be large as well, thus making the layout visually cluttered and less readable. Figure 4.4 shows an example of such a scenario. Therefore, an important objective is to use certain techniques such that the visualization is less cluttered. In the following sections, we discuss two important techniques that help satisfy this objective: *node ordering* and *edge bundling*.



*(a)* *(b)*

***Figure 4.2*** *Balloon Tree Layout*

*. (a) A balloon tree layout of a random graph. (b) Adjacency Edge visualization over the balloon tree layout leads to a cluttered visualization.*

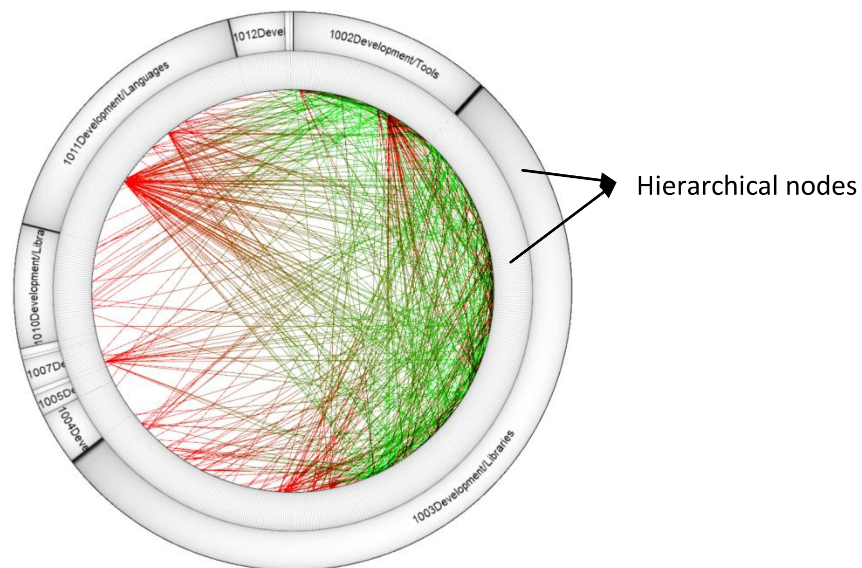
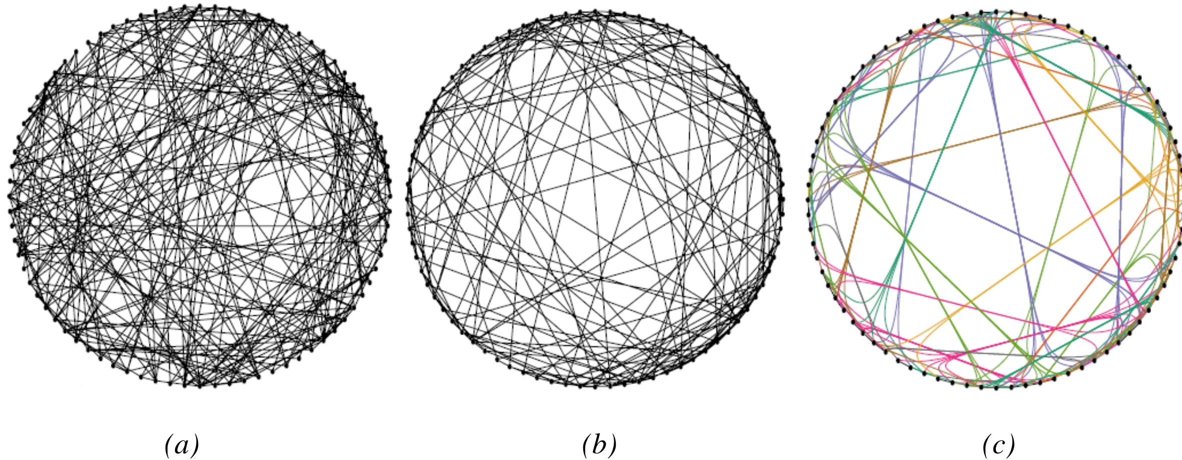



***Figure 4.3*** *ExtraVis Circular View*

*– The visualization shows the hierarchy as well as the dependencies between the nodes.*



(a) (b) (c)

***Figure 4.4*** *Circular Graph Improvements*

*. A Circular graph* (*|V |* = 80, *|E|* = 241). *(a) Random ordering of nodes. (b) Nodes ordered using the dynamic node ordering algorithm. (c) Circular layout of the graph using Edge Bundling [9].*

# 4.3 Node Ordering

To reduce the clutter in circular layouts, we seek to minimize the edge length of the graph. The reason we intend to do this is because longer edges are harder to follow and prone to crossings, thus leading to visual clutter [7, 9, 10]. Therefore, in order to minimize the edge lengths, we re-order the nodes on the circle so that neighboring nodes are closer to each other.

In the following subsections, we describe an algorithm, taken from [9] to find the best ordering of nodes on a circle. Furthermore, we extend this algorithm to address the hierarchical structure of the dataset. Since, the problem of arranging nodes on a circle uniformly such that the angular edge length is minimized is an NP-hard problem, the algorithm is based on a heuristic approach that cannot guarantee finding an optimal solution. The algorithm uses a two-phase approach for obtaining a circular layout with low edge length. The first phase is the *median iteration* followed by *local refinement,* which is the second phase.

## 4.3.1 Median Iteration

The median iteration phase is based on the idea that any node is best situated at the median of its adjacent neighbors in the graph. The position of the median is the median position of the neighbors. First, the nodes are arranged uniformly on a unit circle, after which the median iteration procedure begins. The pseudo code for the median iteration is given below. The $Median_Iteration$ function proceeds by computing, for each node $i$, it's $new_position_i$ by determining the median position of its neighbors. This median is identified using the $Median$ function. Finally, the positions of the nodes are updated using the $updatePosition$ function.

The algorithm results in the nodes moving approximately to the center of their adjacent neighbors. This brings nodes closer to their adjacent neighbors thereby reducing the edge lengths between the nodes. The complexity of a single iteration is $O(|V| + |E|)$. The number of required iterations is generally $n$.

**function Median_Iteration** $(V, E)$

$\%neighbors_i = \{v_j \mid \{v_i, v_j\} \in E, v_i \neq v_j\}$

**repeat** $n$ **times**

**for each** $v_i \in V$ **do**

$new_position_i = Median(neighbors_i)$

**end for**

updatePosition()

**end repeat**

**end**

### 4.3.2 Local Refinement

The median iteration is a simple and fast way for providing a node order with reduced edge lengths. It addresses the global structure of the ordering.

Local refinement considers the local characteristics of the node ordering problem i.e. the edge length cost resulting from the median iteration phase can be further reduced by performing local refinements. This step is performed by considering the set of nodes as a linear arrangement. That is, the nodes are first arranged on a line, and then reordered to further reduce the total edge length.

Let $V$ indicate the set of linearly arranged nodes. The local refinement phase proceeds by first dividing $V$ into three disjoint subsets, namely $L, S, R$ where $L \cup S \cup R = V$. $L$ is the set of nodes placed to the left of $S$, and $R$ is a set of nodes placed to the right of $S$. Figure 4.5 illustrates this arrangement.

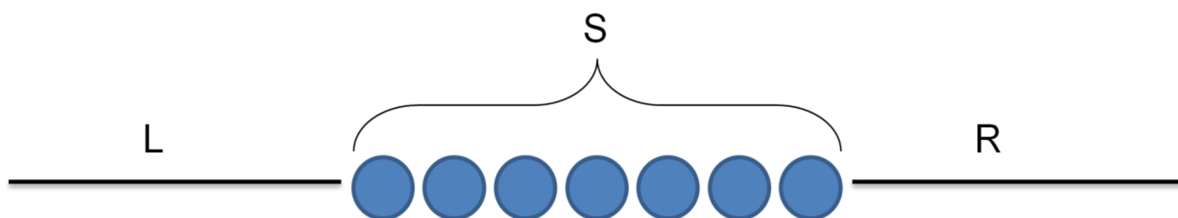



***Figure 4.5*** *A linear arrangement of nodes*

*. The node set is divided into three disjoint sets L, S and R.*

The principle behind the local refinement approach is based on a locality property which implies that when one constructs a node order by incrementally trying all partial arrangements of a $S \subset V$, the best ordering of the remaining nodes does not depend on the exact ordering of the nodes that have already been placed. More formally, taken from [10]

**Theorem 1:** *Let $G(V,E)$ be a graph and $L,S \subset V$ where $|L| = l, |S|, = s$.*
*Let $p, q$ be two orderings of $V$, such that*

$$p(L) = q(L) = \{1, \dots, l\}$$
$$p(S) = q(S) = \{l+1, \dots, l+s\}$$

*Denote by $p_s^*$ the ordering that minimizes the linear arrangement cost, over all permutations that differ from $p$ only with respect to the places of vertices in S. (I.e. for all $i \notin S, p(i) = p_s^*(i)$ .)*

$$q_s^*(i) = \begin{cases} q(i) & i \notin S \\ p_s^*(i) & i \in S \end{cases}$$

*Then, $q_s^*$ minimizes the linear arrangement cost, over all permutations that differ from $q$ only with respect to the places of vertices in S.*

Therefore, among all different orderings of some $S \subset V$ on places $\{1, \dots, s\}$, the best order of $S$ is independent on the orderings of the nodes outside $S$. The proof of the theorem is provided in [10]. In the next subsection, we describe an algorithm taken from [9] based on this theorem to determine the best arrangement of a set of nodes placed on a circle.

#### 4.3.2.1 The MinCA_DP Algorithm

The *MinCA_DP* algorithm is a heuristic approach based on Theorem 1 to find the best ordering of a set of nodes on a circle. Since the theorem is based on a linear arrangement of nodes, we first represent our circular node arrangement a linear arrangement. We do this by determining the $L, S$ and $R$ sets of the circular arrangement. More formally,

$$S = \{v_i, v_{i+1}, v_{i+2}, \dots, v_{i+k} : k \geq 2\}$$

$$L = \{v \in V \backslash S : d(v, v_i) \leq d(v, v_{i+k})\}$$

$$R = \{v \in V \backslash S : d(v, v_{i+k}) < d(v, v_i)\}$$

where $d(v_i, v_j)$ is the angular distance between $v_i$ and $v_j$.

That is, nodes that are closer to the position of the first node of $S$ are said to be in set $L$ while the nodes closer to the position of last node of $S$ are said to be in set $R$. Figure 4.6 shows an illustration of $L, S$ and $R$.

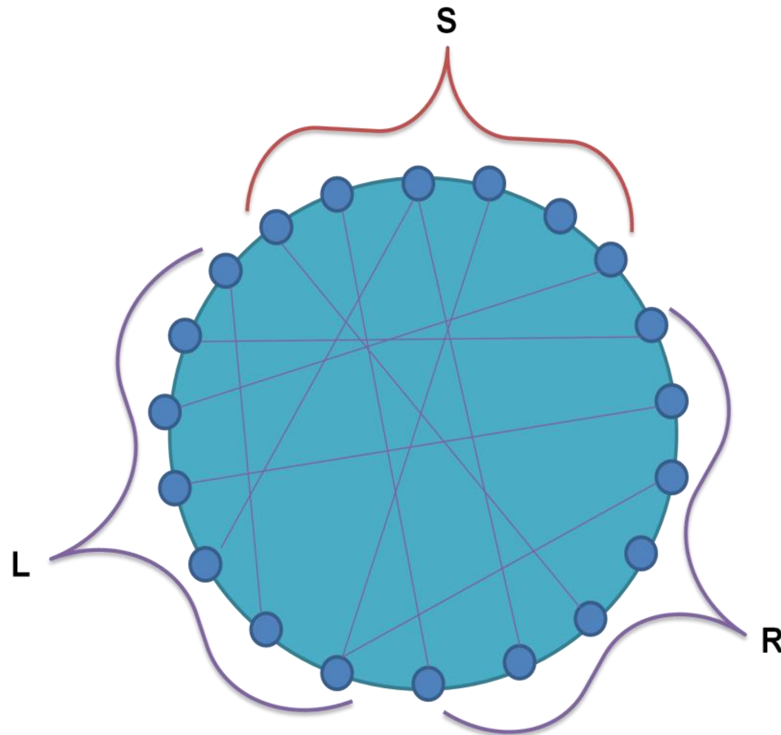



***Figure 4.6*** *Division of nodes on a circle*

*The nodes are divided into disjoint sets in order to represent them linearly.*

In this way, we obtain a linear representation of the nodes on a circle. Once we have the linear arrangement of the nodes, we proceed to determine the best arrangement of the nodes. We define the goal of the algorithm in the following subsection.

**Goal of the MinCA_DP Algorithm**

The goal of the algorithm is to minimize the total edge length between the nodes. The reasoning behind this is the fact that long edges are difficult to follow and more prone to crossings. Therefore, shorter edges can improve the quality of the visualization by reducing the clutter.

More formally,

*For a set of nodes $V$, let $\pi$ be a permutation of $V$. Let $\pi(i)$ be the position of node $i$ in $V$.*

*If $d(i,j)$ is the angular distance between two nodes as position $i$ and $j$, then, the total edge length (Cost of the ordering) is given by*

$$cost_{\pi}(V) \stackrel{\text{def}}{=} \sum_{\{i,j\} \in E} d\,(i,j)$$

The goal of the algorithm is to find an ordering of $V$ that minimizes $cost_{\pi}(V)$.

**The Dynamic Programming Algorithm**

Since the problem of arranging nodes such that total edge length is minimized is an NP-hard problem, [9] gives a dynamic programming algorithm which is based on theorem 1. The algorithm proceeds by finding the best ordering of each overlapping set of $k$ consecutive nodes such that $cost_{\pi}(V)$ is minimized.

Let $\bar{V} \subset V$ be a set of $k$ consecutive nodes $\{v_i \ldots . v_{i+k}\}$. The $MinCA_DP$ algorithm determines the best ordering $p_{\bar{V}}$ for each overlapping $\bar{V} \subset V$ such that cost of each $\bar{V}$, given by $cost_{\pi}^{\bar{V}}(V)$ is minimized. $cost_{\pi}^{\bar{V}}(V)$ is called the *local arrangement cost* of $\bar{V}$. The pseudo code for the overlapping iterations is given below.

**function** $node_ordering(V, E, \bar{V})$

**for** $i = 1$ to $|V|$ **do**

$p_{\bar{V}} \leftarrow MinCA_DP\ (V, E, \bar{V})$

**end for**

**end**

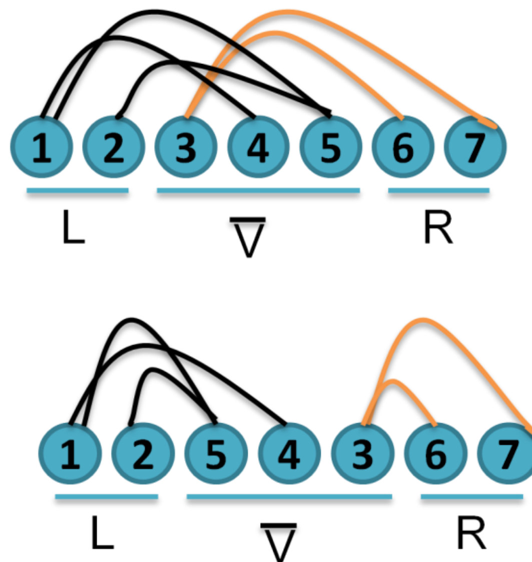



***Figure 4.7*** *Node Ordering of* $V^{-}$

. *(a)* $\bar{V}$ *without node ordering. (b)* $\bar{V}$ *after node ordering such that* $cost_{\pi}^{\bar{V}}(V)$ *is minimized.*

We use Figure 4.7 to illustrate the definition of the local arrangement cost, defined as $cost_{\pi}^{\bar{V}}(V)$. There is a set of linearly arranged nodes $V$ and the relationships between them. The set $V$ is divided into three disjoint sets $L, \bar{V}$ and $R$. The numbers within the nodes indicate the position of the node in the arrangement.

The local length of an edge for a given $\bar{V}$ is the length of the part of the edge over $\bar{V}$. For example, in Figure 4.8a, the local length of the edge $\langle 1,5 \rangle$ is 3 since the edge passes over $\{3,4,5\} \in \bar{V}$. Similarly, the local length of the edge $\langle 1,4 \rangle$ is 2 since the edge passes over $\{4,3\} \in \bar{V}$. We define the *local arrangement cost* of $\bar{V}$ as the sum of the local lengths of all the edges passing over $\bar{V}$.

If $l_{ij}$ is the local length of an edge $\langle i,j \rangle$ passing over $\bar{V}$, then local arrangement cost of $\bar{V}$ is

$$\sum_{\langle i,j \rangle \in E} l_{ij}^{\bar{V}}$$

In our example in Figure 4.7a, the local arrangement cost is

$$cost_{\pi}^{\bar{V}}(V) = l_{\langle 1,4 \rangle}^{\bar{V}} + l_{\langle 1,5 \rangle}^{\bar{V}} + l_{\langle 2,5 \rangle}^{\bar{V}} + l_{\langle 3,6 \rangle}^{\bar{V}} + l_{\langle 3,7 \rangle}^{\bar{V}}$$

$$= 2 + 3 + 3 + 2 + 2 = 12$$

The $MinCA_DP$ algorithm finds an arrangement of $\bar{V}$ such that $cost_{\pi}^{\bar{V}}(V)$ is minimized

In the $MinCA_DP$ algorithm, the $cost_{\pi}^{\bar{V}}(V)$ is defined as a function of cuts, where the cut of a node $i \in \bar{V}$ is given by,

$$cut_i = |\{\{u,v\} \in E: \pi(u) \leq i < \pi(v), i \in \bar{V}, |\{u,v\} \cap S| = |\{u,v\} \cap V \backslash S| = 1, \}|$$

The algorithm defines $cost_{\pi}^{\bar{V}}(V)$ as

$$cost_{\pi}^{\bar{V}}(V) = \sum_{i \in \bar{V}} left(i) + \sum_{i \in \bar{V}, i \neq \pi(v_k)} cut_i$$

where $left(i) = |\{\{i,j\} \in E: i \in \bar{V}, j \in L\}|$ , and $\pi(v_k)$ is the position of the last node in $\bar{V}$.

The $left(i)$ summation takes into account the edges between the $\bar{V}$ and the $L$ while $cut_i$ considers the edges between $\bar{V}$ and $R$ and also edges within $\bar{V}$.

The cost of the complete arrangement $\pi$ of $V$ is given by,

$$cost_{\pi}^{V}(V) = cost_{\pi}^{L}(V) + cost_{\pi}^{\bar{V}}(V) + cost_{\pi}^{R}(V)$$

The proof of the above cost function is given in [10].

Using the algorithm's definition of $cost_\pi^{\bar{V}}(V)$, we have for Figure 4.8a,

$$cost_\pi^{\bar{V}}(V) = \sum_{i\in\bar{V}} left(i) + \sum_{i\in\bar{V}, i\neq\pi(v_k)} cut_i = 3+5+4 = 12$$

Similarly,

$$cost_\pi^{L}(V) = \sum_{i\in L} left(i) + \sum_{i\in L, i\neq\pi(v_l)} cut_i = 0+2 = 2$$

$$cost_\pi^{R}(V) = \sum_{i\in\bar{V}} left(i) + \sum_{i\in\bar{V}, i\neq\pi(v_r)} cut_i = 2+1 = 3$$

Therefore,

$$cost_\pi^{V}(V) = cost_\pi^{L}(V) + cost_\pi^{\bar{V}}(V) + cost_\pi^{R}(V) = 12+2+3 = 17$$

The *MinCA_DP* algorithm attempts to find the best ordering of a set $\bar{V}$ by determining the best ordering of all the partial arrangements of $\bar{V}$ in an incremental manner. The pseudo code of this step is given in Figure 4.9.

For the example in Figure 4.7a, the algorithm finds the best ordering of $\bar{V}$ by first determining the local arrangement costs of its smallest subsets {3},{4},{5}. The algorithm uses the resulting costs to find the local arrangement costs of the subsets of the next higher size. Finally using these results, the ordering of {3,4,5} which has the least local arrangement cost is derived. The algorithm determines the best arrangement by comparing the local arrangement costs. That is, for each partial arrangement, the algorithm identifies the best right most node of the arrangement such that the cost is minimized. For example, for a set {3,4}, the algorithm identifies the best right most node of this set by comparing the costs of the arrangement {3,4} and {4,3}, where cost {3,4} is computed using the previous cost of {3} and cost of {4,3} is computed using the previous cost of 4. If the arrangement cost of {4,3} is less than {3,4}, then right most node of {3,4} is {3}. The procedure continues for every increasing subset $S$ until $|S|$ to $|\bar{V}|$.

Using the illustration in Figure 4.7a,

$$cost_\pi^{\{3\}}(V) = 3, cost_\pi^{\{4\}}(V) = 3, cost_\pi^{\{5\}}(V) = 3$$

$$cost_\pi^{\{3,4\}}(V) = \sum_{i\in\bar{V}} left(i) + cut_3 = 3+2 = 5$$

$$cost_\pi^{\{4,3\}}(V) = \sum_{i\in\bar{V}} left(i) + cut_4 = 3+0 = 3$$

Since, $cost_{\pi}^{\{4,3\}}(V) < cost_{\pi}^{\{3,4\}}(V)$, the right most node of {3,4} is 3. Similarly, the algorithm determines the right most node of every subset of $\bar{V}$, incrementally thus identifying the best ordering of $\bar{V}$.

The algorithm reiterates this process for every overlapping set of $k$ nodes. The ordering is true in the global sense according to theorem 1 and therefore does not depend on the orderings of the nodes that have already been placed. The complexity of the algorithm is $O(n(2^k + |E|))$.

**function** $MinCA_DP(\bar{V} \subset V, ordering)$

% $Cut(i, S)$ is the number of edges between $i$ and the elements of $S$

$left(i) = \{\langle i, j\rangle \in E \mid d(j, v_1) < d(j, v_k), j \notin \bar{V}\}$Type equation here.

**for every** $S \subset \bar{V}$ **do**
  $table[S].cost \leftarrow \infty$
**end for**
  $table[\emptyset].cost \leftarrow 0$
  $table[\emptyset].cut \leftarrow= \sum_{i \in \bar{V}} |left(i)|$

% Fill table:
**for** $i = 1$ to $k$ **do**
  **for** every $S \subset \bar{V}, |S| = i - 1$ **do**
    $cut^s \leftarrow table[S].cut$
    $new_cost \leftarrow table[S].cost + cut^s$ % total edge length is a sum of cuts

    **for** every $j \in \bar{V} - S$ **do**
      **if** $table[S \cup \{j\}].cost > new_cost$ **then**
        $table[S \cup \{j\}].cost \leftarrow new_cost$
        $table[S \cup \{j\}].right_{vtx} \leftarrow j$
      $table[S \cup \{j\}].cut \leftarrow \sum_{x \in S \cup \{j\}} cut_x$

      **end if**
    **end for**
  **end for**
**end for**

S = $\bar{V}$
**for** $i$=$k$ **to** $1$ **do**

  $v = table[S].right_{vtx}$

  $ordering[i] = v$

$S = S - \{v\}$

**end for**

**end**

***Figure 4.8*** *MinCA_DP node ordering*

### 4.3.2.2 The MinCA_DP Algorithm for hierarchies

The *MinCA_DP* algorithm described in the previous subsection finds the best ordering of nodes on a single level. Since the dataset we use is a hierarchical structure, we recursively determines the best arrangement of each level of the hierarchy by using this algorithm. The pseudo code of the recursion is given below.

**function** $node_ordering(V, E, \bar{V})$

**for** $i = 1$ to $|V_{level}|$ **do**

$p_{level} \leftarrow$ MinCA_DP ($V_{level}$ , $E_{level}$ , $\bar{V}_{level}$ )

**end for**

**if** not leaf level **then**

ArrangeChildren($p_{level}$ , $V_{level+1}$)

$node_ordering(V_{level+1}, E_{level+1}, \bar{V}_{level+1})$

**end if**

**end if**

The algorithm proceeds by finding the ordering of each level of the hierarchy in a top down manner. After the nodes of a given level are ordered, the child nodes are arranged such that the restriction of the hierarchy is maintained. The *ArrangeChildren* function does this by taking the permutation of the parent nodes $p_{level}$ . The process continues recursively for each level.

Relationships between higher level nodes depend on the relationships of their leaf children. The strength of the relationship between two higher level nodes is also dependant on the leaf children. We define the strength of a relationship to increase with the number of relationships between the leaf children.

The current implementation of the *MinCA_DP* finds the best arrangement of the higher level nodes without considering the strength of the relationship between these nodes. Figure 4.9 shows an illustration of this problem. It can be seen that even after the best arrangement is identified, groups *Application/Text* and *Application/Productivity* are relatively further apart from each other despite having a large number of relationships between their leaf children. Since the relationship is based on the relationships between the leaf children, an adequate ordering of the hierarchy can be

achieved if we consider the strength between the higher level nodes. In the next subsection, we modify the *MinCA_DP* algorithm to accommodate this fact.

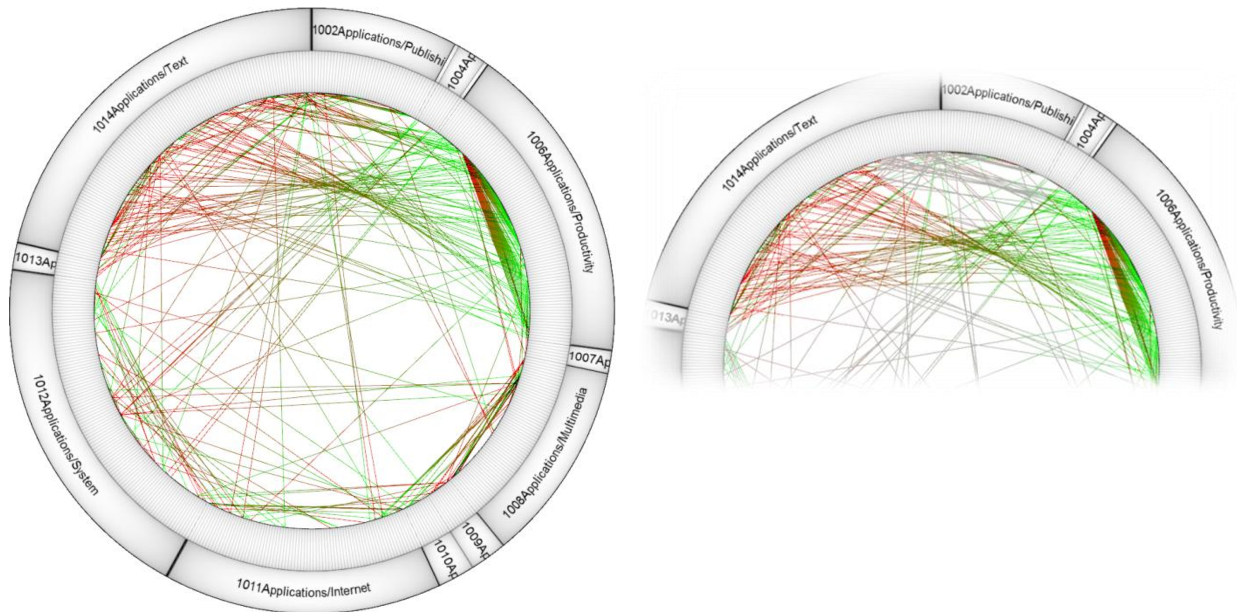



***Figure 4.9*** *Hierarchical node ordering problem*

*. Even after node ordering, higher level nodes connected by large number of edges are farther from each other.*

**4.3.2.2 The Weighted MinCA_DP Algorithm for hierarchies**

In order to place strongly related higher level nodes closer to each other, we modify the *MinCA_DP* algorithm such that the strength between the nodes is considered. We do this by adding a weight to the edges connecting higher level nodes. We use the number of child relationships between the two higher level nodes as the weight of the edge between them. The modification to the algorithm is made by modifying the cost function of the *MinCA_DP* algorithm.

That is, we define the cost as

$$cost_{\pi}(V) \stackrel{\text{def}}{=} \sum_{\{i,j\}\in E} w_{ij} \, . \, d_{ij}$$

where, $w_{ij}$ is the number of edges between the children of $i$ and $j$. If $i$ and $j$ are leaves, $w_{ij}$ is simply equal to 1.

Similarly, the local arrangement cost of each $\bar{V} \subset \mathrm{V}$ is modified by computing adding weights to the cuts.

$$cut_i = \sum_B w_{ij}$$

where B = $\{u, v\} \in E: \pi(u) \leq i < \pi(v), i \in \bar{V}, |\{u, v\} \cap S| = |\{u, v\} \cap V \backslash S| = 1$

Therefore,

$$cost_{\pi}^{\bar{V}}(V) = \sum_{i \in \bar{V}} left(i) + \sum_{i \in \bar{V}, i \neq \pi(v_k)} cut_i$$

where $left(i) = \sum_{i \in \bar{V}, j \in L, \{i,j\} \in E} w_{ij}$, and $\pi(v_k)$ is the position of the last node in $\bar{V}$.

Applying the above modifications to the *MinCA_DP* algorithm results in a better hierarchical ordering by placing strongly related higher level nodes closer to each other. Comparisons on the results achieved by the algorithm are discussed in the next chapter.

# 4.4 Edge Bundling

Another method of improving the quality of circular bundling technique is by using Edge Bundling. The principle of the technique is the controlled deformation of edges, such that groups of edges share common segments. This arrangement can help improve the utilization of the drawing area, giving a cleaner and more readable visualization. Figure 4.4(c) shows the resulting visualization on using the edge bundling technique.

Figure 4.2(b) shows the visualization of a hierarchy using tree visualization. There are inclusion relationships between groups and packages and also dependency relationships between groups at a higher level and between packages at the lower level. It can be observed from the figure that visualizing dependency relationships on top of the tree visualization, leads to visual clutter.

To alleviate this problem, the Hierarchical Edge Bundling technique in [5] can be used. The main advantages of this technique are:

- This technique can be used in conjunction with existing tree visualization techniques and facilitate the integration of existing tools.
- Hierarchical Edge Bundling helps reduce visual clutter when dealing with large number of dependency edges.
- It provides a continuous way to control the strength of the bundling. Low bundling strength provides lower level detail such as node to node connectivity information. High bundling strength provides higher level relationships which is a result of the dependency edges between their respective child nodes.

Figure 4.10(b) shows an example of hierarchical edge bundling. Figure 4.10(a) shows the same visualization without edge bundling.

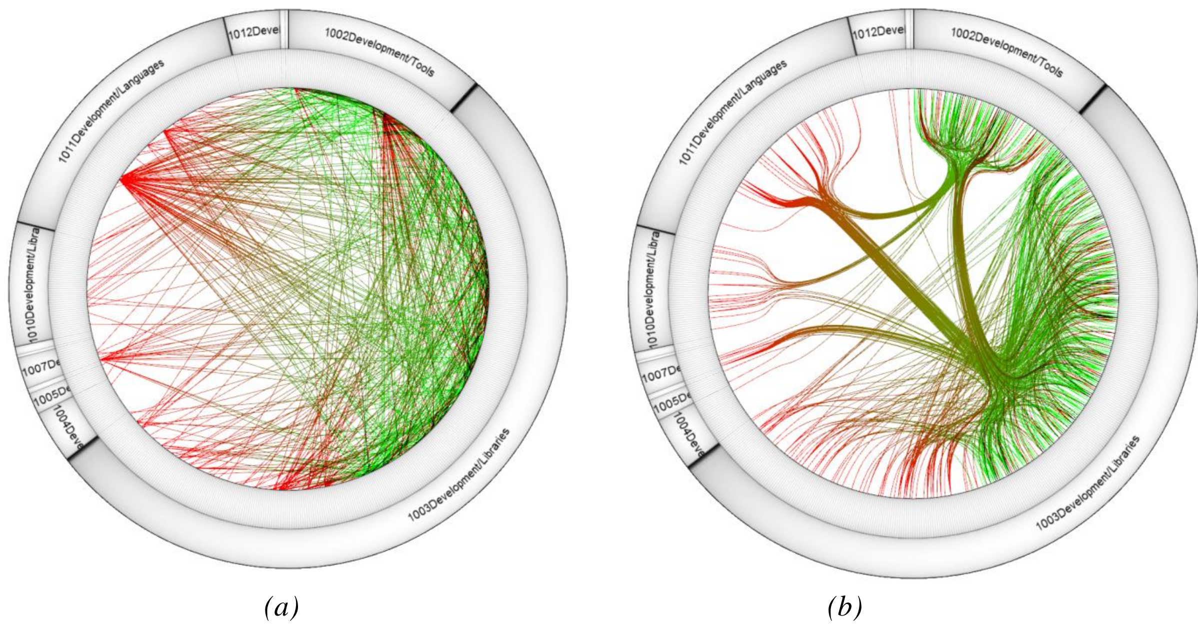



***Figure 4.10*** *Edge Bundling*

*(a) Hierarchical circular layout visualization without edge bundling. (b) Hierarchical circular layout visualization with edge bundling. The gradient edges show the direction of dependency.*

Chapter 5

# Experimental Results

In the previous chapter, we visualized package relationships using circular layouts. In order to improve the visualization, we used edge bundling and node ordering algorithms. In this chapter, we evaluate the node ordering algorithm and its output for our package data set. We perform three different experiments on the visualization with respect to node ordering:

1) Visualization with random node ordering.
2) Visualization with hierarchical node ordering.
3) Visualization with weighted hierarchical node ordering.

We use the package dataset from the Linux Fedora 10 installation CD. The dataset consists of 2220 packages, 13 groups and 60 subgroups. The number of relationships between the packages is 12486. We use the ExtraVis [11] tool to visualize our scenarios. The comparison between the node-ordering techniques is based on two criteria namely, total edge length, and total number of edge crossings. The evaluated algorithms are coded in table 5.1. From the table, we observe that node ordering techniques improve the visualization significantly with reduced edge crossings and edge length. Moreover, the weighted hierarchical node ordering algorithm further reduces the edge length and crossings. Therefore, we see a significant improvement in using node ordering algorithms for circular layouts. The time taken for reordering the complete system was found to be 6 minutes and 29 seconds.

Figure 5.1 shows the visualization output of each node ordering scenario. The visualization shows the relationship between the packages of the *Application* group. Figure 5.2 shows boxed areas of Figure 5.1. The areas chosen in the figure are simple to indicate the general number of crossings and thus, do not consider the positions of the nodes. It can be noticed that the random node ordering in Figure 5.2(a) has more edge crossings compared to that of the other node ordering algorithms. Weighted hierarchical node ordering algorithm results in the best visualization as seen in Figure 5.2(c).

| Scenario | Edge Crossings (in millions) | Edge Length (in millions) |
|---|---|---|
| **Random Node Ordering** | 3.85 | 5.73 |
| **Hierarchical Node Ordering** | 3.42 | 5.70 |
| **Weighted Hierarchical Node Ordering** | 2.74 | 5.60 |

***Table 1****: Comparison of node ordering techniques*

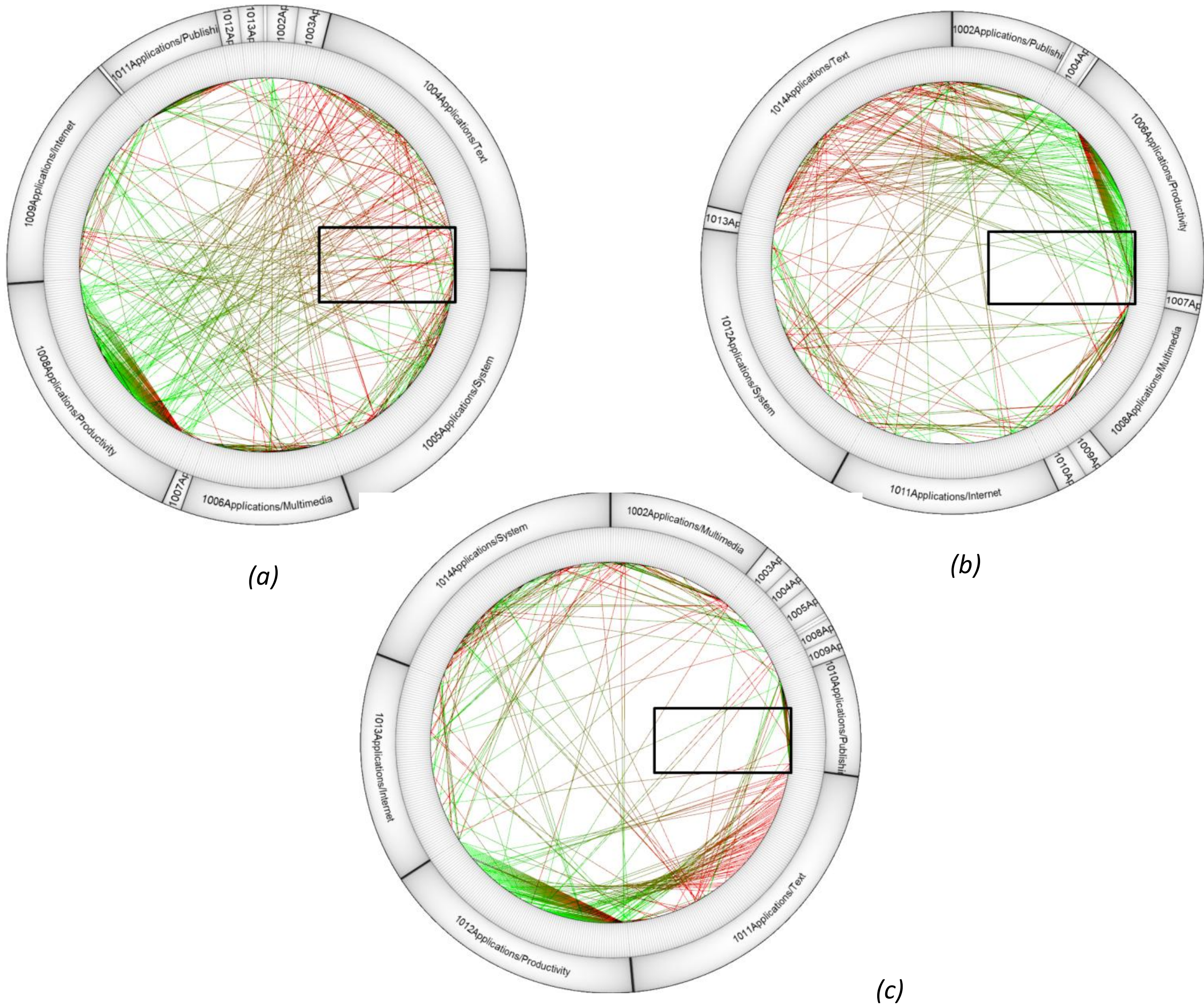
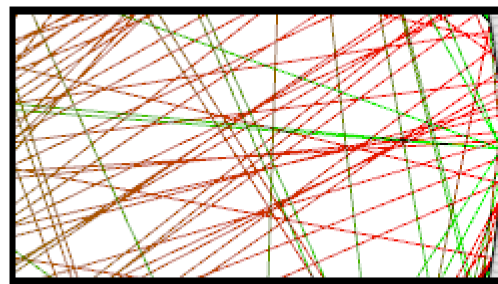
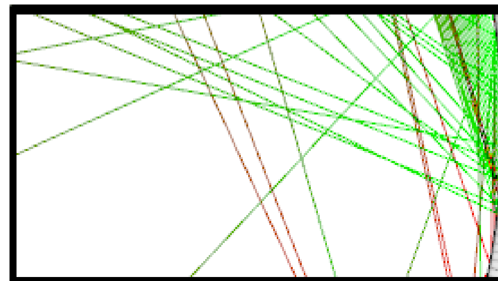



***Figure 5.1*** *Node Ordering Visualization Comparison*

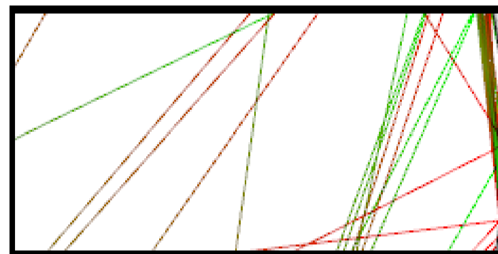
. *V=504, E=600 (a) Random Node Ordering. (b) Hierarchical Node Ordering. (c) Weighted Hierarchical Node Ordering.*



*(a)*



*(b)*



*(c)*

***Figure 5.2*** *Zoomed Node Ordering Comparison*

*. (a) Random Node Ordering has larger edge lengths and hence more crossings. (b) Hierarchical Node Ordering. (c) Weighted Hierarchical Node Ordering reduces edge lengths and thus reduces clutter.*

Figures 5.3 and 5.4 demonstrate the improvement in the visualization from hierarchical node ordering to weighted hierarchical node ordering. The weighted algorithm used in Figure 5.3(b) reorders higher level nodes such that nodes with more number of child relationships to each other are placed closer. This helps reduce the total length of the edges. It can be noticed in Figure 5.4(a) that without the weighted ordering, nodes are placed farther even though they have more number of relationships between them. On the other hand, the visualization in Figure 5.4(b) after using the weighted algorithm, has lesser edge lengths, and thus, lesser clutter.

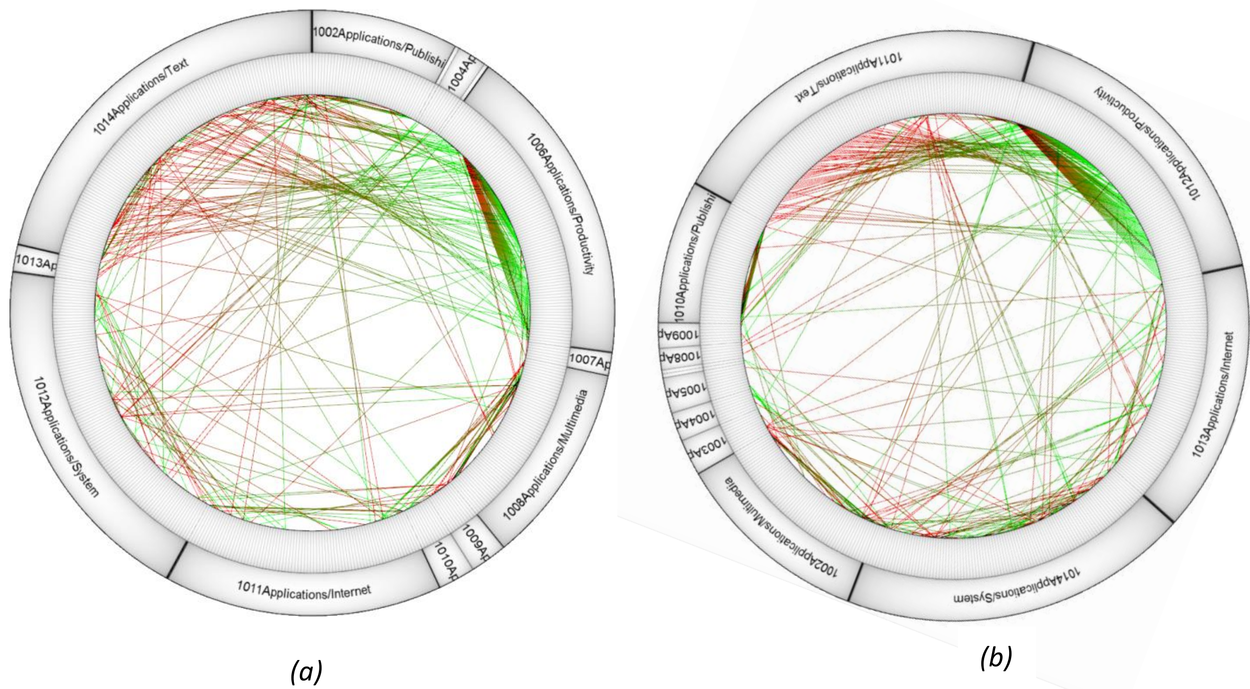



***Figure 5.3*** *Hierarchical Node Ordering*

*. (a) Hierarchical Node Ordering. (b) Weighted Hierarchical Node Ordering brings strongly related higher level nodes closer to each other thereby reducing the total edge length.*

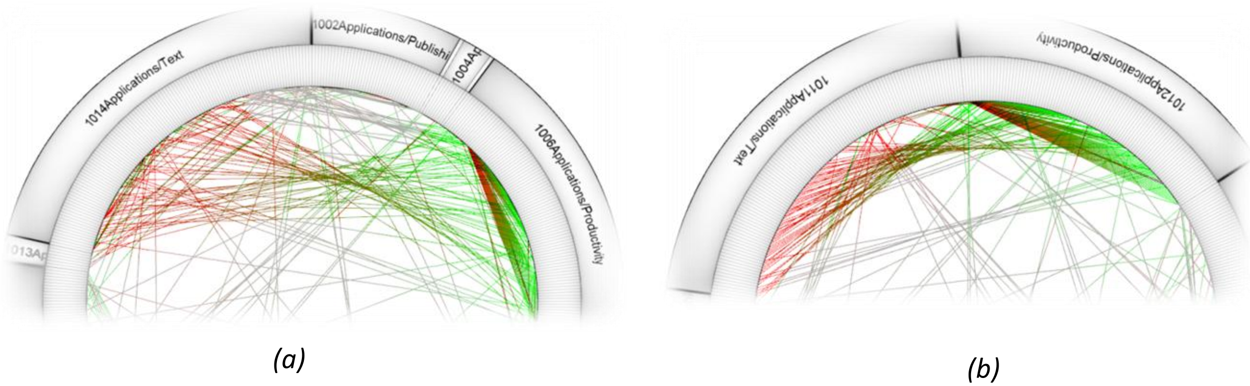



***Figure 5.4*** *Zoomed Hierarchical Node Ordering*

*. In hierarchical node ordering (a), strongly related higher level nodes are placed farther from each other resulting in longer edges as compared to (b) where the weighted algorithm results strongly related higher level nodes being placed closer to each other thus reducing the edge lengths.*

Although the node ordering algorithms help improve the visualization by reducing edge lengths and edge crossings, it does so at the cost of detail. When edges become too short, they become less visible and sometimes completely invisible. Figure 5.5a shows this situation. Edge bundling alleviates this problem by elongating the edges such that they are visible. Also, bundling of edges shows relationships between higher levels of the node hierarchy. This information is not available when the bundling technique is not used.

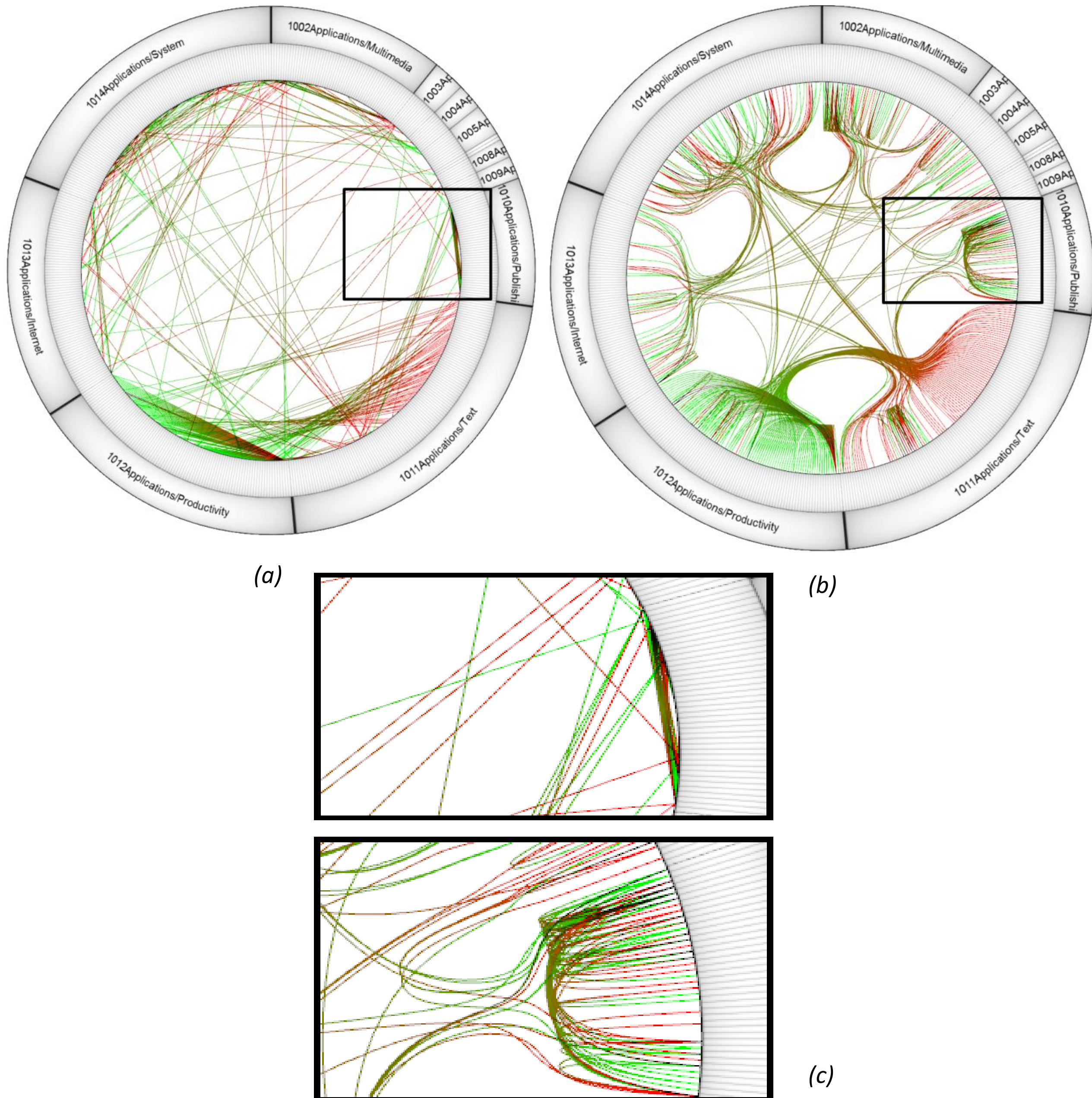



***Figure 5.5*** *Edge Bundling Benefits*

*. (a) No Edge Bundling reduces visibility of shorter edges. (b) Edge Bundling elongates edges. c) Zoomed images of the boxed areas.*

Figure 5.6 shows node ordering being used with edge bundling. It can be noticed that with node ordering, the thick bundle at the diameter of the layout in (a) is reduced by moving the two nodes closer to each other. Also, the clutter within each group is reduced on applying the node ordering technique. Therefore, node ordering provides a further improvement in the visualization after the edge bundling technique is used.

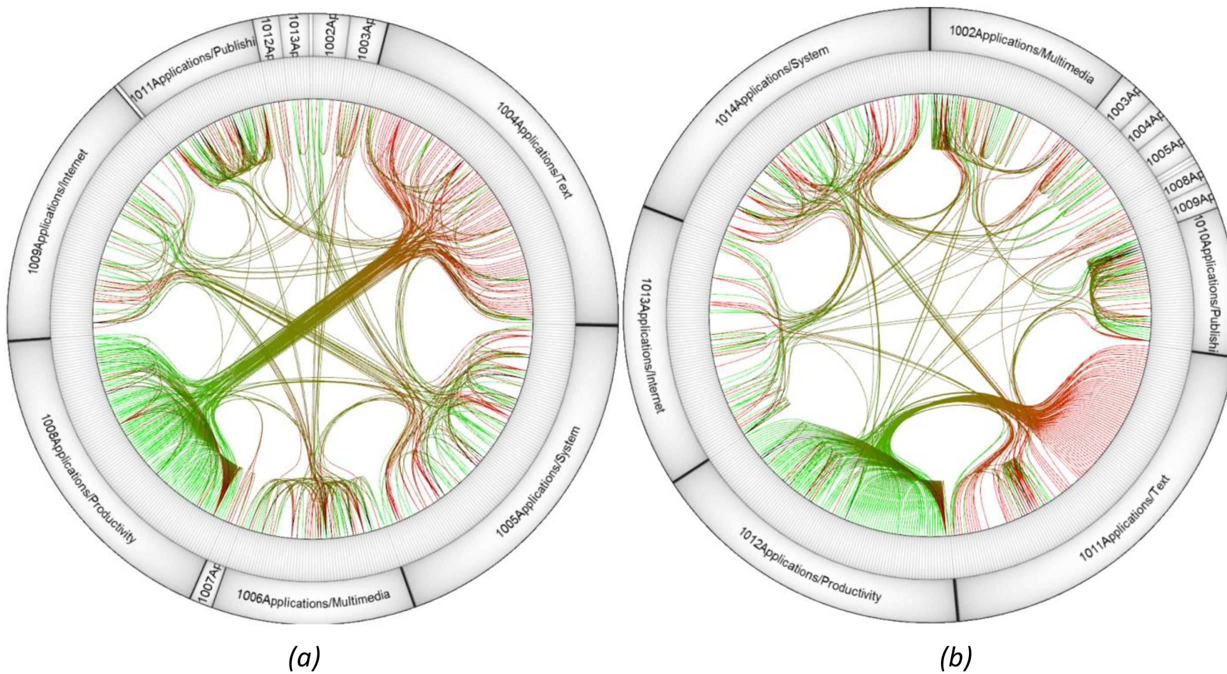



***Figure 5.6** Edge bundling with node ordering*

*. (a) Edge bundling without node ordering. (b) Edge bundling with node ordering.*

Chapter 6

# Conclusions and Future Work

This chapter we summarize the thesis project and derive conclusions from our experiments. Section 6.2 provides suggestions to future possibilities in further improving the visualizations.

## 6.1 Conclusions

Since the information provided by the current RPM software is mostly textual, we constructed a visualization tool so that users could see the contents of the system in a graphical form. The tool enabled users to see specific relationships between packages, groups, the member packages of groups, sub-groups and even capabilities. The tool also enabled users to see the impact of modification on the system, for example, the removal of a package. The tool was also highly interactive and users could control the visualization as desired.

Although our tool had the ability to visualize packages conveniently, visual clutter was introduced when the number of relationships increased. In order to address this issue, we decided to use circular layouts for our visualization because of its regularity and ability to visualize large number of relationships in a reasonable time with lesser clutter. We also worked to further improve the quality of the circular layout visualization by using edge bundling and node ordering techniques. These techniques helped improve the visualization by making better utilization of the display space and improving the readability of the relationships.

The node ordering algorithm that we used was defined for single level circular layouts. Since our dataset was hierarchical in structure, we extended the node ordering algorithm to include the hierarchy. We found that, although on a single level, the algorithm produces impressive results, the introduction of the hierarchy caused the layout to become more restrictive thus restricting the order of the nodes. Despite this rigidity, we managed to achieve satisfactory results and the resulting visualization thus obtained gave a clear and convenient overview of the dataset.

Despite the fact that the node ordering algorithms we used gave us a cleaner picture, it is important to note that this is not always the best picture. From our study, we understand that the definition of

a best picture depends on the perception and needs of the user. The node ordering algorithms we used improved the general presentation of the visualization. However, for the analysis of different behavior of the system, it is important to note that a different node ordering may help a different cause. For example, if the user would want to see the relationship patterns between packages with respect to the size of the packages, then an ordering based on the size of the packages would be the recommended way to go. However, to reduce the ink used to print the graphs, and efficiently utilize the display space, the node ordering algorithms that we have discussed in this report provide satisfactory results.

## 6.2 Future Work

The visualization tool that we developed was advantageous in visualizing smaller number of relationships but as the number of relationships grew larger, the visualization performance deteriorated. Therefore, we used a circular layout and this provided satisfactory results in visualizing the dataset as a whole. However, we analyzed each of these layouts separately because they were not integrated together as a single tool. Having circular layouts integrated with force directed layouts would make our tool more powerful and convenient. Moreover, the reordering algorithm that we used to extend the ExtraVis tool was done by modifying the input file. It would be desirable to implement this reordering tool within our tool itself, by adding options to reorder nodes based on different criteria. For example, size of a package, or creation date, etc. Currently our visualization tool is implemented for Linux packages. A step forward would be to enable it to visualize any other system with properties and relationships similar to our dataset.

# References


1. **Jeffrey Heer, Stuart K. Card** and **James A. Landay** (2005), *Prefuse: A Toolkit for Interactive Information Visualization*.

2. **Edward C. Bailey**. *Maximum RPM*. Red Hat, Inc. 2000.

3. **Ugur Dogrusoz, Brendan Madden** and **Patrick Madden**. *Circular Layout in Graph Layout Toolkit*. Tom Sawyer Software, Berkeley, CA 94710.

4. **Thomas M. J. Fruchterman** and **Edward M. ReinGold**. *Graph Drawling by Force-directed Placement.* Software – Practice and Experience. 1991.

5. **Danny Holten**. *Hierarchical Edge Bundles*: Visualization of Adjacency Relations in Hierarchical Data. IEEE Transactions on Visualization and Computer Graphics. 2006.

6. **Brett McLaughlin**. *Java and XML Data Binding*. First Edition O'Reilly. 2002.

7. **T. Lengauer** and **R. Mijller**. *Linear Arrangement Problems on Recursively Partitioned Graphs*. Zeitschrift for Operations Research, Volume 32, page 213-230.

8. **W.T. Tutte** (1962). *How To Draw A Graph*.

9. **Emden R. Gansner** and **Yehuda Koren**. *Improved Circular Layouts*. M. Kaufmann and D. Wagner page 386 – 398. 2007.

10. **Yehuda Koren** and **David Harel**. *A Multi-Scale Algorithm for the Linear Arrangement Problem*. Department of Computer Science and Applied Mathematics. The Weizmann Institute of Science, Rehovot, Israel.

11. **Bas Cornelissen, Danny Holten, Andy Zeidman, Leon Moonen, Jarke J. van Wijk,** and **Arie van Deursen –** *Understanding Execution Traces Using Massive and Circular Bundles*.

12. **Jean-Yves Mengant**. *Writing a Java Class to Manage RPM Package Content* – Linux Journal. http://www.linuxjournal.com/article/3462

13. **Ulrik Brandes, Markus Eiglsperger and Jurgen Lerner**. *GRAPHML Primer*. http://graphml.graphdrawing.org/primer/graphml-primer.html

14. **Nick Urbanik**. *RPM and Yum.* Free Software Foundation.