EDOS deliverable WP2-D2.1: Report on Formal Management of Software Dependencies

Roberto Di Cosmo

Deliverable WP2-D2.1

Report on Formal Management of Software Dependencies

Project Acronym	Edos
Project Full Title	Environment for the Development and Distribution of Open Source Software
Project number	FP6-IST-004312
Contact Author	Roberto Di Cosmo, roberto@dicosmo.org
Authors List	WP2 Team
Workpackage number	WP2
Deliverable number	1
Document Type	Report
Version	Id: cover.tex 167 2005-09-29 15:00:00Z dicosmo
Date	December 2, 2008
Distribution	Consortium, Commission and Reviewers

Contents

Chapter 1

Introduction

Large software systems are prevalently built in a modular way: different parts of the system are put together in a consistent way, forming a whole which is the final, usable system. This practice is often referred to as *Component Based Software Engineering (CBSE) and Development*, and has been fostered by software engineers because of its intrinsic benefits; building software systems by using *components*, in fact, allows:

- *Reusability*: A single component can be used in different contexts (systems)
- *Maintainability*: By having a system composed of different, loosely coupled components, it is possible to change part of the system itself without interfering with the other (working) parts of the same system.
- *Scalability*: The complexity of extending a software system can be handled by adding and assembling new components in a suitable way.

CBSE has been (and continues to be) a hot research topic in the software engineering field, and it has actually revolutionized the way software is built. Actually, we can find components everywhere in modern software, starting from the operating system drivers, to the plugins we use everyday in order to extend our browsers to make them visualize, for example, pages with rich content such as movies and so on.

The CBSE paradigm has been proven very useful with respect to the Free and Open Source Software (F/OSS) development and distribution process. In fact, a component based approach gives a great help in handling the issues given by the intrinsic heterogeneity of the F/OSS development communities and the decentralized way of building F/OSS software.

Actually a component-oriented approach to build complex software has always been the very nature of the "Unix philosophy". Assembling small and simple utilities together into more complex ones is the usual way Unix users do their everyday tasks.

The definition of the term *component* is very generic and can capture a broad set of software artifacts. A widely accepted definition has been given by Clemens Szyperski[**?**]: *"Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system"*

However, depending on the granularity of the component itself we can differentiate among different component types:

- *Fine grained components*: The previously cited command line tools such as `cat`, `awk`, etc. They are assembled in more complex components by the means of pipes.
- *Plugins*: A software component which is able to extend the functionality of a particular kind of sotware. Plugins are often known also as *modules* or *extensions*. They are deployed and installed in the context of the application which supports them (i.e. a Firefox extension for handling a GMail account, a perl module for the Apache HTTP Server or an Eclipse plugin for editing latex files).
- *Packages*: An assembly which provides a big software unit that is installable in an environment that is typically an operating system.

It is clear that all the previously cited component types have the same abstract purposes and, to some extents, present the same kind of problems. However the more the component is coarse grained, the more the issues to deal with are hard.

Currently, the most widespread component type, in the F/OSS world, is the package. This is due to the fact the mainstream operating system distribution (i.e., Linux based operating system, BSD, etc.) use the abstraction of package to denote an installable software unit, i.e. a component, that contributes to extend the system itself.

However, plugins are gaining importance because of the success of some F/OSS projects such as Eclipse and Firefox that exploited and publicized the idea of using components (called plugins and extensions respectively) in order to extend the functionality of the basic platform.

In the following we will detail the characteristics of those kinds of components, and in particular we will focus on the issues related to their composition. We examine the state of the art and the currently provided solutions to the component handling and assembling, highlighting the pros and cons of each solution. Finally we propose a set of metadata which can further describe the component and help the automated assembling.

Chapter 2

Packages and package management systems

> *A software package is a special method for the distribution and installation of software on computer systems.*
>
> *The most common type of software package seen by the average computer user is that found sold in stores. An example might be a popular word processor. A user would purchase the software, then follow the given instructions to install the software on their home machine.*
>
> *A very common type of software package is that found on many Unix-like operating systems. These are often a single file containing many more files to be installed, along with rules describing what other software needs to be installed for the package to function properly.*
>
> http://en.wikipedia.org/wiki/Software_package

Packages are a convenient way for users to get new (or updated) software installed on their computer. A package is more than just a collection of files to be installed. It usually contains additional information required for the proper installation and/or uninstallation: other packages that this package depends on, directories for files to be installed, menu items for the desktop environments, scripts to be executed before/after installing/uninstalling the package, and more. Packages are usually not installed manually by the user, but using a package manager. The package manager's duty is to automate (as much as possible) the process of installing, upgrading, configuring, and removing software packages from the user's computer.

Packages may be either binary packages or source packages. Binary packages contain the files needed for installation and proper functioning of the software, but not the source files. The files in the binary package are precompiled and so usually are expected to work on a limited set of machine architectures.

Source packages contain the files needed for compilation of the software on the user's computer. The source package contains a compilation script (typically in form of a Unix Makefile) which automates the compilation and (afterwards)

installation processes. It is considered good practice to also enable an uninstallation procedure in the makefile. These source packages, which are originally intended for compilation and (de-)installation of the software on the target machine, are used by F/OSS distributors as basis for their own source packages. The derived distribution-specific source package contains again a compilation script which compiles the source files and arranges all relevant files into bundles constituting the binary packages.

Source packages are more flexible, because the user may choose to tweak the source and because the compilation will usually be optimized for the user's architecture at compile time. However, most users are not expected to be able to cope with software compilation on their machines and to fix problems with the compilation. Also, local compilation can slow down the machine quite considerably.

In this chapter we will present a detailed survey of the current package managements systems and their package formats in order to review what are the currently used metadata and how they are used by package management systems.

2.1 Debian Packages

DEB[**?**], the package management system used by Dpkg, the package manager for the Debian distribution, was created in 1993 by Ian Jackson. It has been upgraded since then, and the format is now in version 2.0.

There are two types of packages: *binary packages* and *source packages*. Binary packages contain files that can be installed directly from the package file; source packages contain source code that can be used to create binary packages—it is possible to create multiple binary packages from one source package.

An example of the contents of a DEB file (the `ocaml` binary package from `debian-unstable`) can be seen in figure 2.1.

2.1.1 DEB file format

A DEB package (binary or source) is an `ar` file which has three members (see figure 2.1): the package version (which nowadays is 2.0) and two `gzip`-compressed `tar` archives containing, respectively, metadata (in proper DEB terminology, the *control data*) and the files that are to be installed as part of the package.

The control archive contains all metadata. Besides a *control file* in which the metadata are stored, it contains MD5 sums for the package data, as well as scripts that are to be run when installing or removing the package.

2.1.2 DEB metadata

Binary packages

The *control file*, which contains all metadata, is a text file which consists of *paragraphs*, each of which consists of *fields*. The paragraphs are separated by blank

- `ocaml_3.08.3-8_i386.deb`
  - `debian-binary` (version)
  - `control.tar.gz`
    - `postinst` (post-install script)
    - `prerm` (pre-removal script)
    - `postrm` (post-removal script)
    - `md5sums` (MD5 sums for `data.tar.gz`)
    - `control` (package metadata)
  - `data.tar.gz`
    - `/usr`
    - `/usr/lib`
    - `/usr/lib/ocaml/3.08.3`
    - `...`

Figure 2.1: Composition of a DEB package

lines.

A field is usually a single line which contains the field name, followed by a colon and the field contents. It is possible to include fields that span multiple lines; in that case, the second and further lines start with a space.

An example of a control file, once again from the `ocaml` package, can be seen in figure 2.2.

A list of all possible fields (except for dependencies) that occur in the control file of a binary package follows:

`Package` **(mandatory)** The package name. This name must consist only of lowercase letters, digits, plus and minus signs and periods. It must be at least two characters long and start with a letter or a digit.

`Source` This field identifies the name of the source package from which the package was created.

`Version` **(mandatory)** The package version number. For a more detailed explanation of version numbers, see below.

`Section` This field can be used to classify packages. There are three sections: `main`, `contrib` and `non-free`. For more on the different sections, see below.

`Priority` Can be one of `required` (the system will not function without it), `important` (the bare minimum expected on any Unix system), `standard`

```
Package: ocaml
Version: 3.08.3-7
Section: devel
Priority: optional
Architecture: i386
Depends: ocaml-base (= 3.08.3-7), ocaml-base-3.08.3,
         ocaml-nox-3.08.3, ocaml-base-nox (>= 3.08.3-6)
Suggests: xlibs-dev, tcl8.4-dev, tk8.4-dev
Provides: ocaml-3.08.3
Installed-Size: 7052
Maintainer: Debian OCaml Maintainers
            <debian-ocaml-maint@lists.debian.org>
Description: ML language implementation with a class-based
 object system Objective Caml is an implementation of the
 ML language, based on the Caml Light dialect extended with
 a complete class-based object system and a powerful module
 system in the style of Standard ML.
 ...
```

Figure 2.2: Example of a DEB control file

(installed by default), `optional` (usually installed but not necessary, for example the X Window System) and `extra` (everything else). Packages on the `optional` level and above should not conflict with each other.

`Architecture` **(mandatory)** The architecture can either be a specific architecture (such as `i386` or `sparc`), or `all` to specify an architecture-independent package.

`Essential` If this field is set to `yes`, the package should not be removed under any circumstances, though it can be upgraded or replaced.

`Installed-Size` The size of the package when it is installed.

`Maintainer` **(mandatory)** The person responsible for the package.

`Description` **(mandatory)** The first line of this field contains a single-line description of the package; a more detailed description in a few paragraphs can be found in the following lines.

Then follow the package dependencies. In the DEB format, there are several different types of dependencies (in the list, we assume that package A depends on package B, i.e. that the dependency for package B figures in the control file of package A):

`Depends` Package B must be configured before package A can be configured. This means a *run dependency*; package A cannot run without package B.

`Recommends` Like `Depends`, but package B is not *absolutely* necessary. However, package B will usually be needed in order for package A to function properly.

`Suggests` Like `Depends`, but package A can function properly without package A.

`Enhances` The opposite of `Suggests`; "package A enhances package B" is similar to "package B suggests package A"

`Pre-Depends` This is an *install dependency*; package A cannot be installed without package B.

`Conflicts` This is the opposite of `Depends`. It is impossible for two conflicting packages to be installed on one system at the same time.

`Replaces` This field provides for a way to resolve conflicts. If package A and B conflict, and if package A replaces package B, then package B will be removed and package A will be installed. Furthermore, this field can also be used to indicate that a package overwrites files from another package (something that would normally lead to an error).

A dependency can also limit the versions of the package that satisfy it. For example, in the `ocaml` package shown above, the version of `ocaml-base` installed must be exactly equal to 3.08.3-7, while the version of `ocaml-base-nox` installed must be greater than or equal to 3.08.3-6.

There are five different 'version operators':

`=` Exactly equal;

`<=` Earlier or equal;

`>=` Later or equal;

`<<` **or** `<` **(deprecated)** Strictly earlier;

`>>` **or** `>` **(deprecated)** Strictly later.

Virtual packages

It is also possible to declare dependencies on *virtual packages*. A virtual package is a package that does not in itself exist, but must be *provided* by another package. For example, the `ocaml` package provides the virtual package `ocaml-3.08.3`; any dependency on `ocaml-3.08.3` can be satisfied by the `ocaml` package.

A more complex example would be a virtual package named `web-server`. A package that needs a web server, but is not interested in any particular web server, could declare a dependency on `web-server`. Any package that provides `web-server` could then satisfy that dependency.

Virtual packages do not have versions, but the possibility to add this functionality to later versions of the DEB format is specifically left open.

In the `ocaml` example, however, we can see that some notion of versions has already been informally added: the `ocaml` package depends on the virtual package `ocaml-3.08.3`, which is provided only by `ocaml` version 3.08.3; earlier versions provide `ocaml-3.08.2` and so on, in effect providing some sort of version requirement.

2.1.3 Source packages

As referenced several times earlier, the DEB format also has source packages. From one source package, it is possible to build several binary packages, and this is reflected in the fact that the control file for a source package consists of one general paragraph and several paragraphs for the binary packages that can be created from the source package.

The control information of a source package is different from the one used in binary packages: A source package may *build*-depend on or *build*-conflict with other packages, thus expressing requirements for the source package to compile. Since the source package is in general common to several architectures it contains schemata for the control information of the binary packages which are instantiated at compilation time. For instance, the *architecture* may now either consist of a *list* of architectures (where `any` abbreviates the list of all supported architectures), or `all` for an architecture-independent binary package. Dependencies in the schema for the control information of a binary package may be qualified by an architecture specifier. The schema may also contain *variables* which are instantiated at compilation time.

2.1.4 DEB version numbers

A DEB version number consists of three components:

First, the *epoch*, a single integer number. This is the most important component; whatever the rest of the version number, a package of epoch $n+1$ will always be of higher version than a package of epoch n. It is intended to be used in case of a change in version numbering scheme, or if a mistake is made. The epoch is optional (if not present, epoch 0 is assumed), and separated from the upstream version by a colon. If there is no epoch, the upstream version may not contain any colon.

Then, the `upstream version`. This usually is the original version of the software that has been packaged. It may contain letters, digits, periods, plus and minus signs and colons. It should start with a digit.

Next comes the optional `Debian revision`, separated from the upstream version by a minus sign; if the Debian revision is not present, the upstream version may not contain a minus sign. The Debian revision, which is of the same form as the upstream version, indicates the version of the Debian package based on the upstream version; therefore, it changes if the Debian package is changed, but the upstream version is not. It is conventional to reset the Debian revision to at 1 every time the upstream version is changed.

Version comparison is done from left to right; first the epoch, then the upstream version and finally the Debian revision are compared.

For any two strings that must be compared (epoch to epoch, upstream version to upstream version or Debian revision to Debian revision), firstly the initial parts that contain only non-digit characters are determined and compared lexicographically. If there is no difference, the initial parts of the remainders that contain only digits are compared numerically. This process (comparing non-digit strings lexicographically and digit strings numerically) is repeated until either a difference is found or both strings are exhausted.

2.1.5 DEB sections

The `main` section

The packages in the `main` section all comply with the Debian Free Software Guidelines[**?**]. Furthermore, packages in the `main` section do not have any 'positive' dependencies (`Depends`, `Recommends` or `Build-Depends` dependencies) on packages outside the `main` distribution.

They also conform to a certain standard of quality ("they must not be so buggy that we refuse to support them").

The `contrib` section

The `contrib` section contains packages that do conform to the Debian Free Software Guidelines, but that do not satisfy the requirement of having no dependencies on packages outside the `main` section.

The standard of quality is the same as for the `main` section.

The `non-free` section

In the `non-free` section, packages can be placed that do not conform to the Debian Free Software Guidelines.

Non-US sections

Each of the three sections mentioned above has a non-US subsection. Packages that are in the `main` section cannot depend on packages that are in the non-US subsection of `main`, but it is possible for packages in the non-US subsection of

`main` to depend on packages that are in the `main` section. The same goes for the `contrib` section.

2.2 RPM Packages

In this section we detail the format of RPM packages [**?**], starting from their structure and focusing particularly on the attributes which details the metadata associated with the package and, in particular, attributes which represent the relationships with other packages. RPM packages can be of two different types:

- *Binary* packages: contains a compiled and ready to install/run packages software.
- *Source* packages: contains the source code to build and package the software into a binary package.

In this document we will address only binary packages.

2.2.1 RPM package naming convention

RPM packages follow a well defined naming convention in order to maintain consistency between the name of the package file and the information encoded in the RPM package format and contained in the file itself. This naming convention is also endorsed by all the tools that supports RPM package creation. Since all the information regarding the RPM package are self contained in the package itself, an RPM package will continue to be usable even if its file name is renamed to some other file name which does not follow the naming convention.

Moreover, it is important to notice that the same naming convention is also used in some metadata fields in the RPM package format (see Section 2.2.2)

The standard RPM package naming convention is the following:

`name-version-release.architecture.rpm`

where:

- `name` is the name of the packaged software (e.g., `bash`, `xorg`, `gnome`, etc.). Often package names are used for describing *subpackages*, that is, packages which derives from the same software distribution. This is the case, for example, when a single software is split into different packages, each one of them providing different and additional functionalities. For example, the `xorg-x11-libs` software package provides all the libraries needed to run `X11` applications, while `xorg-x11-libs-devel` provides also the files needed to compile and link applications against `X11` libraries.
- `version` is the packaged software version. It may contain any character except the dash (’`-`’) one.

- `release` is usually a number which indicates how many times the software has been packaged. However it is usually used also to give other kinds of information, such as the initials of the packaging entity (e.g., `mdk`). The `release` field follows has the same restrictions of the `version` one.

- `architecture` is a string describing the hardware architecture name the package has to be run on. The string `noarch` is used when a given package is compatible with all the architectures (e.g., packages which contains software written in some scripting language). On the other hand, The string `src` can be used instead of the actual architecture name in order to indicate that the package is *source package* which actually contains the source code to build the software. Table 2.1 shows the list of the supported architectures.

Here are some examples of package names found in various Linux distributions: `mc-4.6.1-0.pre3.2mdk.i586.rpm`, `gedit-0.9.7-2.i386.rpm`, `gaim-1.3.0-1.fc4.i386.rpm`, `kphone-4.1.1-1.fc4.x86_64.rpm`.

Notice how the `release` field is often used, besides to show the actual release number, to indicate the distribution the package belongs to as well (i.e., Mandriva/Mandrake (`mdk`), Fedora Core4 (`fc4`), etc.)

2.2.2 RPM file format

RPM packages are bundled in a binary format. The format is the same for both binary and source packages. The current version of the RPM format is 3.0 and it is used by all the RPM package managers since version 2.1.

An RPM package is divided into four logical sections:

- *Lead*: Contains the package format signature and some information concerning the structure of the package itself.

- *Signature*: A collection of digital signatures that are use to sign the package by using cryptographic techniques.

- *Header*: Contains all the package metadata, such as package description, package relationships etc.

- *Payload*: The actual archive which contains all the files that are bundled with the package.

Being the RPM a (multi-platform) binary file format, it has been designed in order to be correctly handled by the RPM package manager, despite of the actual platform it is executed. In particular the reference byte-ordering is the one defined for the Internet (network byte order)

The `Lead` section

The `Lead` section of an RPM package is basically used as a "signature" in order to identify the file as an RPM package. For example, the Unix `file`[1] command uses this information in order to recognize the format. RPM package managers and other RPM oriented tools use this information as well.

Much of the information that is present in the `Lead` section is obsolete and is actually ignored by current RPM package managers. Moreover, that information is duplicated in the `Header` section. It is maintained only for backward compatibility of the file format with older tools.

The structure of the `Lead` section is represented by the data structure `rpmlead`[2] and is made of the following fields:

- `magic` (4 bytes)
  The byte sequence `0xED 0xAB 0xEE 0xDB` which uniquely identify the file as an RPM package.

- `major`, `minor` (1 byte each)
  Two bytes representing the major and minor version of the RPM package format. The current version available, at the time when this document has been written, is `3.0`.

- `type` (2 bytes)
  Two bytes representing the type of the RPM package. Currently only two RPM package type are provided: *binary* (`type == 0`) and *source* (`type == 1`).

- `archnum` (2 bytes)
  Two bytes representing the hardware architecture the RPM package has been built for. The actual architecture is however indicated in the `Header` section. Table 2.1 shows the supported architectures and their mappings to the `ID` used in the `archnum` field.

- `name` (66 bytes)
  A null-terminated, zero-padded, string representing the name and the version of the package using the standard RPM package name specification conventions (see Section 2.2.1).

- `osnum` (2 bytes)
  Two bytes representing the Operating System the package was built for. Table 2.2 shows the supported operating systems and their mappings to the `ID` used in the `osnum` field.

[1] The `file` command tries to associate the format/type to the file which is passed as its argument
[2] Defined in `lib/rpmlib.h` in the RPM source tree [**?**]

- `signature_type` (2 bytes)
  Two bytes representing the format of the `signature` section. Currently, version 3.0 of the RPM package format mandates the *header-style* format for this section and, therefore, the value of this field will always be set to `5`.

Architecture	ID	Architecture	ID	Architecture	ID
i386	1	alphaev6	2	ppc64	16
i486	1	sparc	3	ppc64iseries	16
i586	1	sun4	3	ppc64pseries	16
i686	1	sun4m	3	m68k	6
Athlon	1	sun4c	3	rs6000	8
Pentium3	1	sun4d	3	ia64	9
Pentium4	1	sparcv8	3	armv3l	12
AMD	1	sparcv9	3	armv4b	12
x86_64	1	sparc64	10	armv4l	12
AMD64	1	sun4u	10	s390	14
ia32e	1	mips	4	i370	14
alpha	2	mipsel	11	s390x	15
alphaev5	2	IP	7	sh	17
alphaev56	2	ppc	5	xtensa	18
alphapcap56	2	ppciseries	5		
alphaev6	2	ppcpseries	5		

Table 2.1: RPM Supported architectures

The `Header` structure

The `header` structure defines the format of the `header` and `signature` section of an RPM package file. The choice of the names is a bit confusing and it is maintained for historical reasons.

The `header` structure is quite complicated and it models a small database where it is possible to store and retrieve arbitrary data by the means of keys, called `tags`.

The `header` structure is composed of several header `entries` that logically provide the actual data. An `entry` is characterized by the following attributes:

- `tag` describes the kind of data that is associated with the current `entry`. Table 2.3 shows the available tags in version 3.0 of the RPM package format.

- `type` defines the type of the data associated with the *tag* for the current `entry`. Table 2.4 shows some of the data types available in version 3.0 of the RPM package format. Currently there are more than 200 tags defined

OS	ID	OS	ID
Linux	1	IRIX64	10
IRIX	2	NEXTSTEP	11
SunOS5	3	BSD_OS	12
SunOS4	4	machten	13
AmigaOS	5	CYGWIN32_NT	14
AIX	5	CYGWIN32_95	15
HP-UX	6	UNIX_SV	16
OSF1	7	MiNT	17
osf4.0	7	OS/390	18
osf3.2	7	VM/ESA	19
FreeBSD	8	Linux/390	20
SCO_SV	9	Linux/ESA	20

Table 2.2: RPM Supported operating systems

Tag	
RPMTAG_NAME	RPMTAG_VERSION
RPMTAG_RELEASE	RPMTAG_EPOCH
RPMTAG_SUMMARY	RPMTAG_DESCRIPTION
RPMTAG_BUILDTIME	RPMTAG_BUILDHOST
RPMTAG_INSTALLTIME	RPMTAG_SIZE
RPMTAG_PROVIDENAME	RPMTAG_REQUIREFLAGS
RPMTAG_REQUIRENAME	RPMTAG_REQUIREVERSION
RPMTAG_CONFLICTNAME	RPMTAG_CONFLICTVERSION

Table 2.3: `Header` structure tags

in the RPM package format[3] and they are used to specify all the metadata information which describe an RPM package.

- `count` defines the number of items that of the specified `type` stored in the actual data associated with the current `entry`. Some of the data types, for example the `STRING` one, allow only a count equal to `1`.

The format of the `header` structure, on the other hand, is made of the following fields:

- `magic` (3 bytes)
  The byte sequence `0x8E 0xAD 0xE8` which identify the beginning of the `header` structure.

- `version + reserved` (1 + 4 bytes)

[3] All the tags are defined in `lib/rpmlib.h` in the RPM source tree [**?**]

Type	ID
NULL	0
CHAR	1
INT8	2
INT16	3
INT32	4
INT64	5
STRING	6
BIN	7
STRING_ARRAY	8
I18NSTRING_TYPE	9

Table 2.4: `Header` structure data types

A byte defining the version of the `header` structure and 4 more bytes reserved for a future usage.

- `entries count` (4 bytes)
  Four bytes representing a 32bit integer which gives the number of entries stored in the `header` structure.

- `data size` (4 bytes)
  Four bytes representing a 32bit integer which gives the total size in bytes of the data associated to all the entries stored in the `header` structure.

- `index` (n * 16 bytes)
  A set of n 16-bytes records, where n is the number of entries specified by the `entries count` attribute, where each record contains the following attributes: `tag`, `type`, `offset`, `count`. These attributes actually defines the logical `entry` described above. The `offset` attribute is a byte offset relative to the beginning of the `data` part of the *header* structure, where the actual data associated with the current `entry` is stored.

- `data` (k bytes)
  A sequence of bytes which contains all the data associated with all the entries stored in the `header` structure. The size of the `data` part depends on the actual data stored for each `entry`.

The `Signature` section

The `signature` section contains one or more digital signatures for assessing the origin of the package. The `signature` section is stored by using the `header` structure format described in Section 2.2.2.

The `signature` section may contain multiple signatures. However every RPM package must have at least a signature which specifies the size of the package

(identified by the tag RPMTAG_SIGSIZE) and a signature which gives the MD5 hash of the package (identified by the tag RPMTAG_SIGMD5).

Multiple cryptographic signatures, identified by the relative tags (e.g., RPMTAG_GPG, RPMTAG_PGP, etc.) could present, but are not required.

The Header section

The header contains all the metadata information regarding the RPM package itself. It is stored by using the header structure format described in 2.2.2 and provides all the information needed to handle a given RPM package.

A detailed description of the relevant metadata attributes is presented in Section 2.3

The Payload section

The payload section contains the actual archive with all the files belonging to the RPM package. The format of the payload is a gzipped cpio archive which is uncompressed, depending on the directives specified in the package metadata, when the package is actually installed.

The format of the cpio archive is SVR4 with a CRC checksum.

2.3 RPM package metadata

In this section we will examine the most relevant package metadata that are present the header section (Section 2.2.2) of an RPM package. In particular we will focus on those metadata describing package relationships with the other RPM packages (i.e., dependency information).

It is clear that all the metadata are encoded using the header structure format described in Section 2.2.2 by means of tags and their associated data. For the sake of clarity, in order to refer to package metadata we will use descriptive names instead of actual tag ID associated to the data.

Moreover, the descriptive names are the ones used in spec files. These are files used by automated packaging tools in order to create RPM packages. A spec file contains all the directive and the metadata information specified in a textual and readable format. Starting from the spec file package tools are able to build a standard RPM package in the format described in Section 2.2.2.

In the following section we will use the syntax and the tags taken from the standard spec file format for describing how to build RPM packages. We will not describe all the directives of the spec file format because this will be out of scope for this report. However it is possible to find a quite complete description of these directives in [**?**]

2.3.1 Descriptive and naming metadata

Descriptive metadata allows the packager to specify informational attributes regarding the package itself. In the following we will detail the most relevant metadata attributes.

Package version format

As already hinted in Section 2.2.1, every package is characterized by a version that is used extensively in package metadata in order to specify relationships between packages.

Generally a complete version specification has the following format:

`[epoch:]version[-release]`

where:

- `epoch` is a monotonically increasing integer which can be omitted and, in this case, it is assumed equal to `0`.
- `version` is an alphanumeric string that cannot contain the dash (’`-`’) character. The version number is usually set by the developer or the upstream maintainer[4]
- `release` has a format similar to the package version and is usually a number that is increased each time a change is made to the package build files.

Obviously package versions impose an ordering on packages which is used when it is necessary to specify package relationships with other packages. The comparison algorithm breaks the package version and is basically a segmented comparison. The version is broken up in segments, each of them containing either alphabetical character or digits. The segments are compared in order, with the rightmost segment being the least significant.

The alphabetical segments are compared using a lexicographical ascii ordering, while the digit segments are compared the same way after having removed any leading zero. If the two digit segments are equal, the longer the bigger.

No additional knowledge is embedded in the algorithm, so a version number `5.6` will be *older* than `5.0000503` because the comparison will be made between `6` and `503` (i.e. `0000503` without leading zeroes), and $503 > 6$.

`Name` tag

The `Name` tag specifies the name of the software being packages. It follows the naming conventions described in Section 2.2.1.

[4] A person who is in charge to build packages for the source base maintained by a third party.

`Version` tag

The `Version` tag specifies the version of the software being packages. Usually it matches the version number of the software itself and it specifies the version part of the complete version specification described in Section 2.3.1.

`Release` tag

The `Release` tag specifies the version part of the complete version specification described in Section 2.3.1.

`Epoch` tag

The `Epoch` tag specifies the `epoch` part of the complete version specification described in Section 2.3.1.

`Description` and `Summary` tags

The `Description` tag is used to provide an in-depth description of the packaged software while the `Summary` tag is used only for giving brief description of the same packaged software.

`Group` tag

The `Group` tag provides a way to organize packages into groups. A group specification consists of a series of strings separated by the '`/`' character. This allows the specification of subgroups as if they were subdirectories
(e.g., `Application/Editors`)

2.3.2 Dependency metadata

Dependency metadata are used to establish relationships between packages. Those relationships are used in order to ensure that once the packaged software is installed, the system will provide anything it needs to work properly (i.e., other packaged software, libraries, etc.)

By (correctly) specifying dependency metadata it is possible to guarantee that package management operations (see Section 2.4) will not break the consistency of the system when they are performed.

In this section we describe the metadata tag that are used in RPM packages in order to explicit relationships between packages, and we will detail their semantics.

Dependency specification

A dependency relation is always specified by using the name of a package and, in case, some additional constraint defined by using arithmetic comparison operator.

This is possible because, as described in Section 2.3.1, version number are totally ordered.

The RPM package format allows the usage of the following comparison operators: `<`, `<=`, `=`, `>=`, `=`, when specifying dependency relation constraints. The semantics of those operators is the standard one, applied to version numbers.

Dependency relation specifications establish relations between the current package in its current version and the set of other packages entailed by the dependency specification:

- If only a package name `P` is specified in the dependency relation, then there is a dependency between the current package in its current version and package `P` in *all* or *any* (depending on the semantics of the relation) of its versions.

- If a package name `P` and a constraint `C` on its version number is specified (e.g., `>= 2.3`) , then there is a dependency between the current package in its current version and package `P` in *all* or *any* (depending on the semantics of the relation) versions that satisfies the constraint `C`

`Requires` tag

The `Requires` tag is used to specify what are the packages that are *needed* in order to make the packaged software work. *At least one* of the packages identified by the dependency specification must be present when installing the current package. Actually (see Section 2.4) the required packages could even not be currently installed in the system. However it is important that during the installation process, in which multiple packages may be processed, there is a package which satisfy the dependency relation. Obviously, if only a single package is requested for installation, then there should be an already installed package which satisfy the dependency relation.

```
Requires:  ncurses, libmpeg >= 2.3
```

The previous example shows a `Requires` dependency where the current package needs whatever version of the package *ncurses* and a version greater or equal to version `2.3` of the package libmpeg installed.

`PreReq` tag

The `PreReq` tag has exactly the same semantics of the `Requires` tag (see Section 2.3.2) but it mandates that *at least one* the packages identified by the dependency constraint must be *already* installed in the system before attempting to install the current package.

`Conflict` tag

The `Conflict` tag is the complement of the `Requires` tag and is used to specify what are the packages that *must not* be installed in order to make the packaged software work. *All* the packages identified by the specification must not be present when trying to install the current package (neither already installed in the system, nor in the package list to be processed)

```
Conflicts:  sendmail
```

The previous example shows a `Conflicts` dependency where the current package cannot coexist with any version of the `sendmail` package.

`Provides` tag (Virtual packages and capabilities)

The `Provides` tag is used to declare a capability which is provided by the current packages. Actually the *provided* capability is often referred to as *Virtual package*, that is actually an alias that can be used in dependency relation to refer to those real packages which provide it.

The `Provides` tag offers also a way to group packages together. In fact, by specifying a dependency relation using a *virtual package* or *capability* identifier, we can implicitly identify all the actual packages which *provide* that *virtual package* through a `Provides` declaration.

Usually, the `Provides` tag is also used to provide file dependencies as if they were virtual packages (e.g., `Provides: /bin/sh`). This is particularly useful when RPM packages are used on systems in part managed by an RPM package management system. The advantage of doing so is that a package providing a virtual package `/bin/sh` can be safely removed without actually removing the file `/bin/sh`.

```
Provides:  lda
```

The previous example shows a package which provides a `virtual package lda`[5]. The identifier `lda` can be used, for example, by the `sendmail` package, in its `Requires` dependency, in order to model the fact that `sendmail` to properly work needs a local delivery agent (`lda`).

It is important to notice that when *virtual packages* are used to specify dependency relations, it is not possible to use version constraint on them. This is obvious because a *virtual package* may be provided by different package types whose version numbers are, of course, incomparable.

Finally, often there are packages that `Provides` a virtual package and Requires the same virtual package. For example the package `bash` provides the virtual package `bash` and also requires it in order to be installed. This could be seen as a contradiction, but for what we have said in Section 2.3.2, the package could be

[5] lda is used as an abbreviation for Local Delivery Agent

nevertheless installed because *bash* will provide all the requirements the package itself.

`Obsoletes` tag

The `Obsoletes` tag which packages are obsoleted by this one. Older versions of the package are automatically obsoleted.

Automatic dependencies

When building a RPM package, a set of dependency relation are implicitly declared. In order to do so, starting from the list of the files that make up the package, for each file in the list the following operations are performed:

- It the file is executable, it is examined by using the `ldd`[6] command in order to find out what are the shared libraries it needs. These shared libraries names are actually added to the RPM package as `Requires` dependencies.
- If the file is a shared library, then its `soname`[7] is added to the capabilities the RPM package provides by using the `Provides` tag.

Even if automatically provided and required library names may seem file names, they are actually capability identifiers that are not related to actual file names contained in the package itself.

For example, a package containing the command `ls` will automatically require the following libraries:

```
linux-gate.so.1
librt.so.1
libc.so.6
libpthread.so.0
/lib/ld-linux.so.2
```

2.3.3 Script metadata

In a RPM package it is possible to find metadata which provide an operational behavior that is executed at some stage, after having issued an operation on a package. These metadata simply specify shell-script or script written in some other interpreted language, and are executed by the RPM package management system.

Many script metadata are used to handle source RPM packages, in order to automate the building process. The following ones, however, are used when action are performed on binary packages, in particular, during installation and removal of RPM packages.

[6] The `ldd`prints the shared libraries required by each program or shared library specified on the command line

[7] The `soname` is the name used by a shared library to determine compatibility between different versions of the same shared library

%pre tag

The %pre script is executed just before the package is to be installed.

%post tag

The %post script is executed after the package has been installed. It usually contain some setup command, such as the execution of `ldconfig` to update the system library cache, or the editing of system wide configuration files (e.g., when a new shell is installed the `/etc/shell` is updated accordingly)

%preun tag

The %preun script is executed just before the removal of a package.

%postun tag

The %postun script is executed just before the removal of a package. It usually contains cleanup code and complementary action with respect to those specified in the %post script.

2.4 RPM package management system

The RPM package management system is build around a single command line utility `rpm`, which provides the user all the functionalities to:

- Build RPM package starting from source code.
- Install RPM packages.
- Remove RPM packages.
- Upgrade RPM packages.
- Query uninstalled RPM packages for information.
- Query the installed base of RPM packages.

`rpm` make use of a central database where it stores all the information about the packages that are already present in the system. Each operation provided by `rpm` query this database in order to perform consistency checks with respect to package dependencies.

2.4.1 RPM package installation

When `rpm` executes a package installation it performs the following steps:

- *Dependency check*: starting from the information provided by package metadata (see Section 2.3.2) `rpm` checks that all the required capabilities and packages are present in the current systems (or in the other packages to be installed with the current one).
- *Conflict check*: if there exists a package which does not satisfy all the conflict constraint described in the package metadata then the installation is aborted.
- *`%pre` script execution*
- *Configuration files processing*: is the package installs some configuration files (as specified in the package metadata) it is possible that these configuration files might overwrite the already present ones. This situation is handled carefully by making backup copies before actually overwrite those files.
- *Payload archive unpacking*
- *`%post` script execution*
- *Central database update*

2.4.2 RPM package removal

When `rpm` executes a package removal it performs the following steps:

- *Dependency check*: `rpm` checks the central database in order to make sure that there is no other package that has a dependency relationship with the package being removed.
- *`%preun` script execution*
- *Config file backup*: if some of the config files have been modified, `rpm` makes a backup copy before removing them
- *File check and removal*: for each file belonging to the package being removed, `rpm` checks if that file belongs also to some other package in the system. If they don't it removes them
- *`%postun` script execution*
- *Central database update*

2.4.3 RPM package upgrade

When `rpm` executes a package upgrade it basically performs first an installation of the package and then a removal of the upgraded ones taking care of correctly handling the various config files that are present in the packages.

2.5 A comparison between the RPM and the DPKG package management systems

Most important features for package management are common to RPM and dpkg (dependencies, versioning, informational metadata, and the like) but certain features are quite different, and we list here the more relevant.

File dependencies File dependencies is a feature that's present in the RPM format but not in DEB. It allows a package to require specific files, instead of packages. The problem is that these dependencies are not explicitly present in the list of provides of the packages, and RPM uses information from the list of files in the packages to handle them. Tools manipulating such dependencies need to find all files required by the packages that they want to install or remove, and look for them in the list of files present in all packages known to the system at execution time. Since this *package universe* changes over time, this kind of file dependencies may become a source of problems for tools trying to perform sophisticated manipulations of package sets.

rpmlib dependencies RPM has some special dependencies for requiring some features that are not present in some versions of RPM itself. They're not provided by the RPM package and are treated as a special case by rpmlib, which individually checks these dependencies against a list of features compiled into rpmlib. Tools that manipulate RPM packages, apart from `rpm` itself, ignore all such features present in the requires list of packages.

ORed dependencies This feature is only present in DEB, but absence of it in RPM does not change the expressibity of the dependency language, as an OR dependency can be directly simulated using an artificial capability and multiple packages providing this very same capability (this is similar to what DEB's `provides:` tag does).

Package relevance DEB files allow to specify some relevance information about packages, that RPM has not

Package priority is an important feature of deb that's absent in RPM. Package priorities tell how important the package is for the system and are used in situations such as when APT needs to choose which package it should install or remove to satisfy some dependency.

Essential packages The essential tag of deb is used by APT to determine whether a package can be removed. If the user attempts to remove a package like glibc or bash, it will issue a warning and ask for confirmation.

Mixed tools like APT-RPM use a file containing a list of all RPM packages and their respective priorities, listing at least the important and essential packages for the distribution being used.

Multiple simultaneously installed versions of a package Debian does not allow two versions of the same packages to be concurrently installed in the system, and APT does not handle that. In that system, packages that are frequently duplicated, such as the kernel or ncurses, are provided with different package names, like ncurses4 and ncurses5.

Architecture variations Some RPM packages have versions compiled with optimizations that are specific to a variation of an architecture. For example, the kernel may have packages compiled for i586 and i686, in addition to the generic i386 package.

2.6 A comparison between the RPM and the DEB package formats

The following table has been taken from [**?**] and shows a comparison matrix between the two package formats DEB and RPM.

Feature	**deb**	**rpm**
Security, authentication, and verification		
signed packages	yes[1]	yes
checksums	yes	yes
permissions, owners, etc	yes	yes
Usability by standard linux tools recognizable by file	yes	yes
data unpackable by standard tools	yes [2]	no [3]
metadata accessible by standard tools	yes	no
creatable by standard tools	yes	no
Metadata		
name	yes	yes
version	yes	yes
description	yes	yes
dependencies	yes	yes
recommendations	yes	no
suggestions	yes	no
conflicts	yes	yes

virtual packages and provides	yes	yes
versioned dependencies and conflicts	yes	yes
boolean package relationships	yes	no [4]
file dependencies	no	yes
copyright info	no [5]	yes
grouping	yes	yes
priority	yes	no
Special files		
config files	yes	yes
documentation files	no	yes
ghost files	no	yes
Package programs		
binary programs allowed	yes	no
pre-install program	yes	yes
post-install program	yes	yes
pre-remove program	yes	yes
post-remove program	yes	yes
verify program	no	yes
triggers	no	yes
Scalability		
no hard-coded limits	yes	yes [6]
new metadata	yes	yes [7]
new section	yes	no
format version data	yes	yes

1. Not yet widely used though.

2. The admin would only have to remember that a deb is an ar archive, containing some tarballs.

3. rpm2cpio can do it, but it's not a standard tool, except on rpm-based systems. Some fairly short programs can do it, but none of them are something you'd want to memorize.

4. An rpm may depend on a list of packages, but boolean OR is not supported. You can often get the same effect using virtual packages and provides. This isn't quite the same, since it does require more coordination between packagers, and the following relationship cannot be expressed with provides: `foo (<< 1.1) | foo (>> 2.0)`

5. Copyright info is included in debian packages, but not in an easily extractable format.

6. Technically, the rpm "lead" contains hard-coded limits on the package name, but the lead is no longer really used by anything except file.

7. To be useful, you need to get a tag number assigned to your new piece of metadata, which implies modifying the rpm program.

The remark made with respect to the boolean package relationships (remark number 4), is not quite exact. In fact, by using the features provided by the RPM package format, *it is* possible, by using the `provides` mechanism, to express boolean OR package relationships. In order to do so, for every distinct OR relationship to be specified we would have to introduce an unique identifier `P` and tag every package participating in the relationship with a `provides P`. However, this solution is so tricky and impractical to implement that it is not a viable alternative in the currently available RPM package management system.

2.7 Ports system

The ports system, as used under various names by FreeBSD, NetBSD (pkgsrc) and Gentoo (portage), is different from the DEB and RPM formats in that it focuses mainly on source packages instead of binary packages; the standard way of installing software is not by installing a binary package, but by compiling it from the original source.

The core of the ports system is a collection of build scripts. In the FreeBSD ports collection and the NetBSD pkgsrc, these are Makefiles; in the Gentoo portage system, they are bash scripts (called "Ebuilds"). These build scripts contain all the instructions for building the software. At the minimum, the build script contains the location where the original source can be found, but there are many possibilities for customization, e.g. the use of the GNU `autoconf` and `automake` programs, patches, specific compiler options, etc.

The ports system (under NetBSD and FreeBSD) consists of a directory tree, with every package having its own directory. These directories contain the Makefiles. In order to install a package, one simply `cd`s to the appropriate directory and types `make install`.

There are minor differences between the ports system as used by FreeBSD, NetBSD and Gentoo, but the basic ideas remain the same. Therefore, we will use the NetBSD pkgsrc system as an example for the rest of this section.

2.7.1 Source packages

Let us look at the NetBSD makefile for the `ocaml` package.

```
# $NetBSD: Makefile,v 1.38 2005/06/14 21:00:41 minskim Exp $

.include "Makefile.common"

CONFIGURE_ARGS+= -no-tk
CONFIGURE_ENV+= disable_x11=yes
```

```
BUILD_TARGET= world
.if (${MACHINE_ARCH} == "i386") || \
    (${MACHINE_ARCH} == "powerpc") || \
    (${MACHINE_ARCH} == "sparc")
BUILD_TARGET+= opt opt.opt
PLIST_SRC= ${PKGDIR}/PLIST.opt
.  if ${OPSYS} != "Darwin"
PLIST_SRC+= ${PKGDIR}/PLIST.prof
.  endif
PLIST_SRC+= ${PKGDIR}/PLIST
.endif

.if ${OPSYS} == "Darwin"
PLIST_SRC+= ${PKGDIR}/PLIST.stub
.endif

.include "../../mk/bsd.pkg.mk"
```

We see that this Makefile invokes another Makefile, called `Makefile.common`. The reason for this is that there is also another package, called `ocaml-graphics`, which provides the OCaML language with support for X11 graphics (the `ocaml` package does not require X11). The common settings for both packages are in `Makefile.common`, whereas the settings that are only for `ocaml` are in the main Makefile seen above.

We see that configure arguments are set in order to disable X11, and that the compilation options are changed depending on the architecture (OCaML native compilation is only available on a few architectures). Also, the Darwin operating systems requires some additional options.

Lastly, the `bsd.pkg.mk` file is included; this is the general file that contains all the code that takes care of automatic downloading, extracting, patching, building and installing.

Here is the common Makefile:

```
DISTNAME= ocaml-3.08.4
CATEGORIES= lang
MASTER_SITES= http://caml.inria.fr/pub/distrib/ocaml-3.08/
EXTRACT_SUFX= .tar.bz2

MAINTAINER= adam@NetBSD.org
HOMEPAGE= http://caml.inria.fr/ocaml/
COMMENT= The latest implementation of the Caml dialect of ML

DISTINFO_FILE= ${.CURDIR}/../../lang/ocaml/distinfo
```

```
PATCHDIR= ${.CURDIR}/../../lang/ocaml/patches

USE_TOOLS+= gmake
HAS_CONFIGURE= yes
CONFIGURE_ARGS+= -prefix ${PREFIX}
CONFIGURE_ARGS+= -libs "${LDFLAGS}"
CONFIGURE_ARGS+= -with-pthread
CONFIGURE_ENV+= BDB_LIBS=${BDB_LIBS} \
BDB_BUILTIN=${USE_BUILTIN.${BDB_TYPE}}
CPPFLAGS+= -DDB_DBM_HSEARCH

.include "../../mk/bsd.prefs.mk"

.if ${OPSYS} == "Darwin" || ${OPSYS} == "Linux"
INSTALL_UNSTRIPPED= yes
.endif

.include "../../mk/bdb.buildlink3.mk"

post-extract: cp-power-bsd cp-gnu-config

cp-power-bsd:
@${CP} ${WRKSRC}/asmrun/power-elf.S ${WRKSRC}/asmrun\
                          /power-bsd.S

cp-gnu-config:
@${CP} ${PKGSRCDIR}/mk/gnu-config/config.guess ${WRKSRC}\
                             /config/gnu/
@${CP} ${PKGSRCDIR}/mk/gnu-config/config.sub ${WRKSRC}\
                             /config/gnu/

.include "../../mk/pthread.buildlink3.mk"
```

Here, a lot more options are set. Firstly, a few options to do with the original source code (the 'distfiles'): how the file is called, where it can be downloaded, and how it can be decompressed.

Also, some information about the software: the person responsible for packaging (the maintainer), the homepage, and a short description ('comment').

Then, the locations for distribution info (the size and MD5 keys of the distfiles, in order to make sure that they have not been corrupted) and patches are explicitly set; the Makefile can be used by different packages, so it could be called from different directories.

Then follow some variables that influence build behaviour; ocaml uses GNU make instead of the standard BSD make, it has a configure script that needs several

arguments, and there are some options that need to be set in order to properly use the Berkeley DB system.

After this, the included `bsd.prefs.mk` once again is a system file that contains code to interpret all these variables.

Then, it is specified that the `strip` utility must not be used under Darwin or Linux.

The `post-extract` target is defined to specify actions that have to be taken directly after decompressing the distribution file(s). There are six phases in the build process, and for each of these, `pre-` and `post-` targets can be defined:

- `fetch`: Download the distribution files.
- `extract`: Decompress the files just downloaded.
- `patch`: Apply any patches.
- `configure`: Configure the build process.
- `build`: Build the software.
- `install`: Install the software.

In the case of the `ocaml` package, some files must be copied after decompressing the distribution files; this is specified under the `cp-power-bsd` and `cp-gnu-config` targets.

Also, the files `bdb.buildlink3.mk` and `pthread.buildlink3.mk` files are included. These files take care of dependencies; apparently, the `ocaml` package needs the Berkeley DB and the pthread libraries.

Here is the buildlink3 file for `ocaml`:

```
# $NetBSD: buildlink3.mk,v 1.12 2005/02/04 21:35:51 adrianp Exp $

BUILDLINK_DEPTH:= ${BUILDLINK_DEPTH}+
OCAML_BUILDLINK3_MK:= ${OCAML_BUILDLINK3_MK}+

.if !empty(BUILDLINK_DEPTH:M+)
BUILDLINK_DEPENDS+= ocaml
.endif

BUILDLINK_PACKAGES:= ${BUILDLINK_PACKAGES:Nocaml}
BUILDLINK_PACKAGES+= ocaml
BUILDLINK_DEPMETHOD.ocaml?= build

.if !empty(OCAML_BUILDLINK3_MK:M+)
BUILDLINK_DEPENDS.ocaml+= ocaml>=3.08.2
BUILDLINK_PKGSRCDIR.ocaml?= ../../lang/ocaml
```

```
. include "../../mk/bsd.prefs.mk"
. if ${OPSYS} == "Darwin"
INSTALL_UNSTRIPPED= yes
. endif

PRINT_PLIST_AWK+= /^@dirrm lib\/ocaml$$/ \
{ print "@comment in ocaml: " $$0; next }

BUILDLINK_TARGETS+= ocaml-wrappers
OCAML_WRAPPERS= ocaml ocamlc ocamlc.opt ocamlcp ocamlmklib ocamlmktop \
ocamlopt ocamlopt.opt

ocaml-wrappers:
${_PKG_SILENT}${_PKG_DEBUG} \
for w in ${OCAML_WRAPPERS}; do \
${SED}  -e 's|@SH@|${SH}|g' \
-e 's|@OCAML_PREFIX@|${BUILDLINK_PREFIX.ocaml}|g' \
-e 's|@CFLAGS@|${CFLAGS}|g' \
-e 's|@LDFLAGS@|${LDFLAGS}|g' \
<${.CURDIR}/../../lang/ocaml/files/wrapper.sh \
>${BUILDLINK_DIR}/bin/$$w; \
${CHMOD} +x ${BUILDLINK_DIR}/bin/$$w; \
done

.endif # OCAML_BUILDLINK3_MK

BUILDLINK_DEPTH:=     ${BUILDLINK_DEPTH:S/+$//}
```

Here, the actual dependency variables are set; the version depended on is ocaml, greater than or equal to 3.08.2.

There are many other variables that can influence the build process in different ways. It is, for example, possible to set compile options both per package and generally (i.e. for multiple packages; the option "ssl" for example will result in all packages being compiled with SSL support, if available).

This system is very flexible, and it is possible, using the Makefile syntax, to specify very complicated build procedures. However, for most packages, specifically those that use a GNU configure script, it is enough to simply specify the location of the source files and a few options.

Virtual packages, as in the DEB and RPM formats, are not supported (possibly because it would require searching through the entire pkgsrc tree in order to find a package that provides the virtual package). They can, however, be emulated (by creating a normal package that depends on the packages that are to provide the virtual package).

2.7.2 Binary packages

This section is based on reverse-engineering.

A binary package, under NetBSD, is simply a tarred and gzipped file that contains the files that are part of the package. Besides that, there are some special files that contain the meta-data:

`+CONTENTS` The files that are installed as part of the package, optionally with MD5 keys.

`+COMMENT` A one-line description of the package.

`+DESC` A longer description of the package.

`+MTREE_DIRS` A description of the directory structure expected by the package.

`+BUILD_VERSION` The CVS tags of all files involved in the building process (in the example, `Makefile` is version 1.38, and `Makefile.common` is version 1.11)

`+BUILD_INFO` Variables involved in the building process, such as compiler flags, architecture, but also dependencies. See below for more information.

`+SIZE_PKG` The size of the package.

The `BUILD_INFO` file

This is arguably the most important file of a binary package, since it contains the package metadata.

The metadata are stored in the form of a list of build variables. Here is an example, again from the `ocaml` package:

```
BDB_TYPE=db1
BDBBASE=/usr
_PLIST_IGNORE_FILES=
_DISTFILES=ocaml-3.08.3.tar.bz2
_PATCHFILES=
PKG_SYSCONFBASEDIR=/usr/pkg/etc
PKG_SYSCONFDIR=/usr/pkg/etc
PKGPATH=lang/ocaml
OPSYS=NetBSD
OS_VERSION=2.0
MACHINE_ARCH=i386
MACHINE_GNU_ARCH=i386
CPPFLAGS= -DDB_DBM_HSEARCH  -I/usr/include
CFLAGS=-O2 -I/usr/include
FFLAGS=-O
```

```
LDFLAGS= -L/usr/lib -Wl,-R/usr/lib -Wl,-R/usr/pkg/lib
CONFIGURE_ENV=BDB_LIBS=  BDB_BUILTIN=yes PTHREAD_CFLAGS=\ -pthread\
 PTHREAD_LDFLAGS=\ -pthread PTHREAD_LIBS= PTHREADBASE=/usr disable_x11=yes
 CC=cc CFLAGS=-O2\ -I/usr/include CPPFLAGS=-DDB_DBM_HSEARCH\ -I/usr/include
 CXX=c++ CXXFLAGS=-O2\ -I/usr/include COMPILER_RPATH_FLAG=-Wl,-R F77=f77
 FC=f77 FFLAGS=-O LANG=C LC_COLLATE=C LC_CTYPE=C LC_MESSAGES=C
 LC_MONETARY=C LC_NUMERIC=C LC_TIME=C
 LDFLAGS=-L/usr/lib\ -Wl,-R/usr/lib\ -Wl,-R/usr/pkg/lib
 LINKER_RPATH_FLAG=-R PATH=/usr/tmp/lang/ocaml/work/.wrapper/bin:
 /usr/tmp/lang/ocaml/work/.buildlink/bin:/usr/tmp/lang/ocaml/work/.gcc/bin:
 /usr/tmp/lang/ocaml/work/.tools/bin:/usr/pkg/bin:/sbin:/usr/sbin:/bin:
 /usr/bin:/usr/pkg/sbin:/usr/pkg/bin:/usr/X11R6/bin:/usr/local/sbin:
 /usr/local/bin:/usr/pkg/bin:/usr/X11R6/bin PREFIX=/usr/pkg
 PKG_SYSCONFDIR=/usr/pkg/etc
 INSTALL_INFO=/usr/tmp/lang/ocaml/work/.tools/bin/install-info
 MAKEINFO=/usr/tmp/lang/ocaml/work/.tools/bin/makeinfo MAKE=make
 WRAPPER_DEBUG="yes" WRAPPER_UPDATE_CACHE="yes"
CONFIGURE_ARGS=-prefix /usr/pkg -libs " -L/usr/lib -Wl,-R/usr/lib
  -Wl,-R/usr/pkg/lib" -with-pthread -no-tk
OBJECT_FMT=ELF
LICENSE=
RESTRICTED=
NO_SRC_ON_FTP=
NO_SRC_ON_CDROM=
NO_BIN_ON_FTP=
NO_BIN_ON_CDROM=
CC_VERSION=gcc-3.3.3
GMAKE=GNU Make 3.80
_PKGTOOLS_VER=20050318
REQUIRES=/usr/lib/libc.so.12
REQUIRES=/usr/lib/libcurses.so.6
REQUIRES=/usr/lib/libm.so.0
REQUIRES=/usr/lib/libm387.so.0
REQUIRES=/usr/lib/libpthread.so.0
```

We see that all kinds of information are stored in the file, including information on dependencies, though the dependencies here are on libraries, not on packages.

Conspicuously absent are the package name and version. These are stored in the `+CONTENTS` file.

2.8 Portage

Gentoo Linux is using a packaging system very different from other distributions. It is inspired by the *BSD ports system*, with new advanced features. This system,

called *Portage* allows to install programs, by compiling them automatically from sources, with all the optimizations for your computer, according to your choices.

More informations about Portage can be found in the *Gentoo Handbook* [**?**], the *Gentoo Developer Handbook* [**?**], and *Gentoo manual pages* [**?**].

Some of the advanced features of Portage are:

- the ability to have multiple versions and revisions of the same package in the tree,
- conditional dependencies between packages,
- sandboxed safe installation,
- configuration file protection and profiles.

Gentoo's release model is based on the following ideas: There is only one package repository, that is evolving continuously. Each package live together with other versions of the same program and you can decide which version you want on your system. Packages are tagged by *keywords*, indicating for each hardware architecture whether it is available, not available, or available but not tested sufficiently. For example, if a package is tagged "`x86 ppc ~alpha -hppa ~amd64`", it means that it is available on x86 and ppc, not available on hppa, not tested sufficiently on alpha and amd64, not tested on other architectures. Packages with a tag "`-`" or "`~`" are *masked*, that means that by default, they won't be installed (but you can decide to override the flag).

Package developers can allow two different versions of a package to be installed in the same system.

2.8.1 The Portage Tree and ebuilds

The *portage tree* is a directory tree on your system where all the informations on packages are stored. There is one directory for each package, containing all the versions of the package. All the informations about a version are in a file called *ebuild*. Ebuilds are bash shell scripts defining variables (`DESCRIPTION`, `HOMEPAGE`, `DEPEND`, `KEYWORDS`, etc.) and bash functions (`pkg_setup`, `src_unpack`, `src_compile`, `src_install`, `pkg_preinst`, `pkg_postinst`, `pkg_config`, etc.), which can use a set of predefined function. Here is an excerpt from an ebuild:

```
# Copyright 1999-2005 Gentoo Foundation
# Distributed under the terms of the GNU General Public License v2
# $Header: /var/cvsroot/gentoo-x86/app-editors/emacs/emacs-21.4-r1.
ebuild, v 1.15 2005/08/23 03:12:54 agriffis Exp $

DESCRIPTION="An incredibly powerful, extensible text editor"
```

```
HOMEPAGE="http://www.gnu.org/software/emacs"
SRC_URI="mirror://gnu/emacs/${P}a.tar.gz
        leim? ( mirror://gnu/emacs/leim-${PV}.tar.gz )"

LICENSE="GPL-2"
SLOT="21"
KEYWORDS="alpha amd64 arm hppa ia64 ppc ppc64 s390 ~sh sparc x86"
IUSE="X Xaw3d gnome leim lesstif motif nls nosendmail"

RDEPEND="sys-libs/ncurses
        sys-libs/gdbm
        X? ( virtual/x11
                >=media-libs/giflib-4.1.0.1b
                >=media-libs/jpeg-6b-r2
                >=media-libs/tiff-3.5.5-r3
                >=media-libs/libpng-1.2.1
                !arm? (
                Xaw3d? ( x11-libs/Xaw3d )
                motif? (
                   lesstif? ( x11-libs/lesstif )
                   !lesstif? ( >=x11-libs/openmotif-2.1.30 ) )
                gnome? ( gnome-base/gnome-desktop )
                )
        )
        nls? ( sys-devel/gettext )
        !nosendmail? ( virtual/mta )"
DEPEND="${RDEPEND}
        >=sys-devel/autoconf-2.58"

PROVIDE="virtual/emacs virtual/editor"
SANDBOX_DISABLED="1"

DFILE=emacs-${SLOT}.desktop

src_unpack() {
...
}

src_compile() {
...
}

...
```

Naming conventions

`pkg-ver{_suf{#}}{-r#}.ebuild` where `_suf` is one of `_alpha` < `_beta` < `_pre` < `_rc` < (no suffix) < `_p` and `-r#` is gentoo specific revision number. For ex `linux-2.4.0_pre10-r2.ebuild`

2.8.2 The USE flags

When you install a Gentoo system, you need to define "USE" flags.

USE variables are used to tell portage:

- what package you want to install
- what features a certain package should support

E.g. if you don't put the kde keyword in your USE flags, packages that have optional KDE support will be compiled without it packages that have optional KDE dependency will be installed without installing the KDE libraries (as dependencies). Default USE is defined in `/etc/make.profile/make.defaults`:

```
USE="oss apm arts avi berkdb bitmap-fonts crypt cups encode
fortran f77 font-server foomaticdb gdbm gif gpm gtk gtk2 imlib
jpeg kde gnome libg++ libwww mad mikmod motif mpeg ncurses nls
oggvorbis opengl pam pdflib png python qt quicktime readline
sdl spell ssl svga tcpd truetype truetype-fonts type1-fonts X
xml2 xmms xv zlib"
```

You can add your own flags in /etc/make.conf, for ex:
`USE="-kde -qt msn yahoo jabber"`

You can declare USE-flags for individual packages (not system-wide), or just for one installation.
List of available USE-flags in `/usr/portage/profiles/use.desc`:

```
gtk     - Adds support for x11-libs/gtk+ (The GIMP Toolkit)
gtk2    - Use gtk+-2.0.0 over gtk+-1.2 in cases where a
                                        program supports both.
gtkhtml - Adds support for gnome-extra/gtkhtml
imap    - Adds support for IMAP
...
```

You can also use *local* USE-flags:

```
app-editors/emacs:multi-tty - Add multi-tty support
app-editors/emacs:nosendmail - If you do not want to install
                                                    any MTA
```

Some packages don't only listen to USE-flags, but also *provide* USE-flags. When you install such a package, the USE-flags they provide is added to your USE setting. (ex: kde provided by kde-base/kdebase)

2.8.3 Dependencies

Dependencies between packages are described in ebuilds in the variables `DEPEND` and `RDEPEND`. `DEPEND` tells Portage about which packages are needed to build the package. The `RDEPEND` variable specifies which packages are needed for the package to run.

In dependencies, you write gentoo packages names:

```
RDEPEND="sys-libs/ncurses
         sys-libs/gdbm"
```

meaning that any version of these packages will fit.

But you can precise a version number, for example:

```
RDEPEND=">=media-libs/giflib-4.1.0.1b
         =media-libs/jpeg-6b-r2
         ~sys-apps/qux-1.0
         =sys-apps/foo-1.2*
         !sys-libs/gdbm"
```

which means that you need a version of giflib newer or equal to 4.1.0.1b, exactly jpeg-6b-r2, (you can also have <, >, or <=), and:

`~sys-apps/qux-1.0` will select the newest portage revision of qux-1.0.

`=sys-apps/foo-1.2*` will select the newest member of the 1.2 series, but will ignore 1.3 and later/earlier series. That is, `foo-1.2.3` and `foo-1.2.0` are both valid, while `foo-1.3.3`, `foo-1.3.0`, and `foo-1.1.0` are not.

`!sys-libs/gdbm` will prevent this package from being emerged while gdbm is already emerged.

As you see in the example, Portage allows to do conditional dependencies. For example `X?` means that the following parenthesis will be in the dependencies only if X is in the USE flags. `!arm?` means that the following parenthesis will be in the dependencies only if arm is not in the USE flags.

A package can also depend on either a package or another one. Examples:

```
DEPEND="|| ( app-games/unreal-tournament
             app-games/unreal-tournament-goty )"
DEPEND="|| ( sdl? ( media-libs/libsdl )
             svga? ( media-libs/svgalib )
             opengl? ( virtual/opengl )
             ggi? ( media-libs/libggi )
             virtual/x11 )"
```

In the last example, one of the packages will be chosen, and the order of preference is determined by the order in which they appear.

Virtuals A package can *provide* a *virtual package*, so that other packages can depend on it. It is useful for example when a package depends on a system logger or a mail transport agent, but not on a particular one.

PDEPEND The variable `PDEPEND` contains a list of all packages that will have to be installed after the program has been compiled.

2.9 Package management systems meta-tools

Dependency management is an important feature of package management systems. It helps keep system consistency, making sure that everything needed for a certain piece of software to work is there, in the expected version.

Tools such as rpm or dpkg have handle dependencies for one single package: they are designed to learn what dependencies a package has and let the user perform an operation affecting the package only when all dependencies are met.

For example, if the user wants to install the nice game known as `frozen-bubble`, she may download from the web the file `frozen-bubble_1.0.0-6_i386.deb`, which contains version 1.0.0, revision 6 of the game.

Then, she may try to install it using the `dpkg` tool, but with little success:

```
dpkg -i frozen-bubble_1.0.0-6_i386.deb
Selecting previously deselected package frozen-bubble.
(Reading database ... 167282 files and directories currently installed.)
Unpacking frozen-bubble (from .../frozen-bubble_1.0.0-6_i386.deb) ...
dpkg: dependency problems prevent configuration of frozen-bubble:
 frozen-bubble depends on libsdl-perl (>= 1.20-8); however:
  Package libsdl-perl is not installed.
 frozen-bubble depends on frozen-bubble-data (= 1.0.0-6); however:
  Package frozen-bubble-data is not installed.
 frozen-bubble depends on fb-music-high | fb-music-low; however:
  Package fb-music-high is not installed.
  Package fb-music-low is not installed.
dpkg: error processing frozen-bubble (--install):
 dependency problems - leaving unconfigured
Errors were encountered while processing:
 frozen-bubble
```

Indeed, `dpkg` checks whether the prerequisites for installing `frozen-bubble` are met, and when this is not the case, it simply fails reporting the missing packages, and urging the user to go and install the prerequisite packages first. This can be a long, annoying and error-prone operation.

Similar scenarios can be found for package removal, since a package cannot be uninstalled until any and all packages that depend on it are removed first.

It's clear that tasks like following the chain of dependencies, dowloading additional packages and solving possible conflicts, can be automated, and that's what package management meta-tools like APT, URPMI and the like are designed to do.

They can install, uninstall, and upgrade packages, automatically handling dependency calculation and package download.

In the example above, the same installation operation of the game `frozen-bubble` can be performed by APT or URPMI as shown below

```
apt-get install frozen-bubble
Reading Package Lists... Done
Building Dependency Tree... Done
The following extra packages will be installed:
  fb-music-high frozen-bubble-data libsdl-console libsdl-gfx1.2
  libsdl-image1.2 libsdl-mixer1.2 libsdl-net1.2 libsdl-perl libsdl-ttf2.0-0
  libsmpeg0
Suggested packages:
  ttf-freefont
The following NEW packages will be installed:
  fb-music-high frozen-bubble frozen-bubble-data libsdl-console libsdl-gfx1.2
  libsdl-image1.2 libsdl-mixer1.2 libsdl-net1.2 libsdl-perl libsdl-ttf2.0-0
  libsmpeg0
0 upgraded, 11 newly installed, 0 to remove and 0 not upgraded.
Need to get 0B/13.5MB of archives.
After unpacking 20.3MB of additional disk space will be used.
Do you want to continue? [Y/n] y
Selecting previously deselected package libsdl-image1.2.
(Reading database ... 87551 files and directories currently installed.)
Unpacking libsdl-image1.2 (from .../libsdl-image1.2_1.2.4-1_i386.deb) ...
Selecting previously deselected package libsdl-console.
Unpacking libsdl-console (from .../libsdl-console_1.3-3_i386.deb) ...
Selecting previously deselected package libsdl-gfx1.2.
Unpacking libsdl-gfx1.2 (from .../libsdl-gfx1.2_2.0.9-4_i386.deb) ...
Selecting previously deselected package libsmpeg0.
Unpacking libsmpeg0 (from .../libsmpeg0_0.4.5+cvs20030824-1_i386.deb) ...
Selecting previously deselected package libsdl-mixer1.2.
Unpacking libsdl-mixer1.2 (from .../libsdl-mixer1.2_1.2.6-1_i386.deb) ...
Selecting previously deselected package libsdl-net1.2.
Unpacking libsdl-net1.2 (from .../libsdl-net1.2_1.2.5-3_i386.deb) ...
Selecting previously deselected package libsdl-ttf2.0-0.
Unpacking libsdl-ttf2.0-0 (from .../libsdl-ttf2.0-0_2.0.6-5_i386.deb) ...
Selecting previously deselected package libsdl-perl.
Unpacking libsdl-perl (from .../libsdl-perl_1.20.3-1_i386.deb) ...
Selecting previously deselected package fb-music-high.
Unpacking fb-music-high (from .../fb-music-high_0.1.1_all.deb) ...
Selecting previously deselected package frozen-bubble-data.
Unpacking frozen-bubble-data (from .../frozen-bubble-data_1.0.0-6_all.deb) ...
Selecting previously deselected package frozen-bubble.
Unpacking frozen-bubble (from .../frozen-bubble_1.0.0-6_i386.deb) ...
Setting up libsdl-image1.2 (1.2.4-1) ...
```

```
Setting up libsdl-console (1.3-3) ...

Setting up libsdl-gfx1.2 (2.0.9-4) ...

Setting up libsmpeg0 (0.4.5+cvs20030824-1) ...

Setting up libsdl-mixer1.2 (1.2.6-1) ...

Setting up libsdl-net1.2 (1.2.5-3) ...

Setting up libsdl-ttf2.0-0 (2.0.6-5) ...

Setting up libsdl-perl (1.20.3-1) ...
Setting up fb-music-high (0.1.1) ...
Setting up frozen-bubble-data (1.0.0-6) ...
Setting up frozen-bubble (1.0.0-6) ...
```

In what follow, we present some of the mainstream automated package management tool.

2.9.1 APT (Advanced Package Tool)

APT (Advanced Package Tool) was initially written by Debian developers (Brian White, Jason Gunthorpe, and contributors) and provides a simple way to retrieve, install and upgrade packages from multiple sources using the command line. Unlike dpkg (.deb format installation tool), APT does not understand .deb files, it works with the packages proper name and can only install .deb archives from a source specified in a configuration file. APT will call dpkg directly after downloading the .deb from the configured sources. There is no apt program per se; APT is a C++ library of functions that are used by several command line programs for dealing with packages, most notably apt-get and apt-cache. More recently a port was written by Conectiva to bring the APT benefits to the RPM based distributions (Conectiva, Red Hat, SuSE, ALT-Linux, etc). There are several front-ends to manage APT more easily, some such as synaptic, gnome-app-install and KPackage take full advantage of a graphical interface making APT even more user friendly.

2.9.2 YUM (Yellow Dog Updater, Modified)

Yum is an automatic updater and package installer/remover for RPM systems. It automatically computes dependencies and figures out what things should occur to install packages. It is written in python and has basically the same functionality as the APT package manager. Its main advantages over the RPM version of APT are its smaller codebase and better dependency handling. A main flaw is its lack

of standard GUIs, although some companies, such as Cobind, have attempted to rectify the problem.

2.9.3 URPMI

URPMI is a meta package manager developed by Mandriva that provides functionalities similar to those found in APT. It allows the user to define media sources for the packages, and is then able to download them from the network, or access a local medium. It uses several heuristics to search for a possible set of packages that allow the installation of the user required packages. URPMI constructs a dependency tree from a set of demanded modules. It begins to load the dependency tree for the known set of packages available in its repositories; then a simple tree-walk algorithm is used to gather all required packages. URPMI being an interactive tool, it is able to propose different sets of packages than can solve the set of requirements for the demanded modules. (For example, to solve a dependency on "webfetch" URPMI can use the "curl" or the "wget" package, so it will ask the user for it.)

The dependencies can be versioned, so URPMI maintains a range of acceptable versions for each dependency, narrowing them down when the tree walk progresses.

When URPMI encounters a conflict (either because it is a conflict explicitly marked in the package, or because two packages A and B require another package C with non-overlapping version requirements), it backtracks in the dependency tree and tries another path.

2.9.4 Smart

Smart [**?**] is a meta package manager similar to the previously cited APT, YUM and URPMI, that provides a set of algorithms which optimize and improve the conflict and dependency resolution problem when trying to install a package.

Smart supports several package formats, notably RPM and DEB, and it allows to setup several package sources by means of *channels*: the Smart's way of model and abstract package repositories.

Smart uses the underlying package management system (i.e., `rpm` or Debian's `dpkg`) in order to perform the actual package installation/removal/upgrade. However it uses the information provided by the package metadata retrieved from the available channels in order to efficiently compute a set of package operations that maximizes a given metric, defined by some in policies.

There are some built-in policies that tries to minimize the impact of a package installation on a stable system, for example, by minimizing the number of needed upgrades.

Smart turns to be an efficient tool because of its ability of exploring and weighting several possible ways of installing a package, including these where the downgrading of some already installed packages would make it possible to install a package that, otherwise, would not be installable at all.

Chapter 3

Remarks on the constraint language

Having seen the global structure of the most used package formats, it is interesting to take a step back, and look at the expressiveness and complexity of the constraint languages we have encountered so far.

This will allow us to establish an upper bound for the algorithmic complexity of the operations that automated tools may be asked to perform on package sets, and get a better understanding of the expressiveness of the constraint language used.

3.1 DEB and RPM both use unary constraints

As we have seen, the basic constraints that are used both in RPM and DEB dependencies may have only one of the following forms:

- P meaning "any version of package P", that can be seen as an abbreviation for $P > 0$
- $P\ op\ const$ where op is a binary comparison operation and $const$ is a constant value, meaning "any version v of package P such that $v\ op\ const$ is true".

This kind of constraint is called *unary constraint* in the Constraint Solving community, and is known to be strictly less expressive than binary constraints (which are as general as n-ary constraints).

3.1.1 Reduction to boolean constraints

It is important to remark that the unary constraints appearing in the dependencies of DEB and RPM can be replaced by an equivalent set of boolean constraints.

- For each available version v of a package P in the repository R, introduce the variable P_v

- Replace each unary constraint by a disjunction of boolean constraints as follows
  - P becomes $P_{v_1} \vee \ldots \vee P_{v_k}$ where $v_1, \ldots, v_k$ are the available versions of P in the repository
  - $P\ op\ const$ becomes $P_{v_1} \vee \ldots \vee P_{v_k}$ where $v_1, \ldots, v_k$ are the available versions of P in the repository that satisfy $v_i\ op\ const$

This encoding is immediate in the DEB format, while it requires some gymnastic using the `provides:` tag in the RPM format.

Of course, the size of the boolean encoding can be bigger than the original problem: if the problem is of size n and the maximum number of versions available for a single package is k, the boolean encoding is $O(kn)$, which represents a less than (or equal than) quadratic blowup.

In the following, we will hence focus on package dependencies represented using boolean constraints only.

3.2 Package installation is NP-Complete

Since the language used to combine these constraints involves disjunctions (either explicit, as in the DEB format, or implicit through the `provides:` tags of the RPM format), conjunctions and negations (through the `conflicts:` tag), one might expect that the CSP problem of checking the installability of a given package P using a repository R is computationally difficult.

This is indeed the case, as we show in what follows.

For the sake of clarity, we will only discuss here the `depends:` and `conflicts:` constraints, and assume that the `provides:` tags have been previously inlined.

More formally, the problem is

Configuration. We call *configuration* of a repository R (which is the set of available packages) an assignment $\alpha : R \rightarrow \{\mathrm{Installed}, \mathrm{Uninstalled}\}$ mapping each package in R to the label $\mathrm{Installed}$ or $\mathrm{Uninstalled}$.

The problem. Deciding whether a given package P is installable, given a repository R, corresponds to finding a configuration α of R that assigns `Installed` to P and such that all the constraints associated to the packages in R mapped to $\mathrm{Installed}$ are satisfied. More formally, the installation problem can be represented

as the language

$$\text{PACKINST} = \Big\{ \langle R, P\rangle \;\Big|\; \exists \alpha \;\; \alpha : R \to \{\text{Installed}, \text{Uninstalled}\} \wedge \\ \alpha(P) = \text{Installed} \wedge \\ \forall P' \;\; \alpha(P') = \text{Installed} \Longrightarrow \text{the constraints associated to } P' \text{ in } R \\ \text{are satisfied in } \alpha. \Big\}$$

where $\langle R, P\rangle$ is a suitable encoding of the repository R and the package name P.

3.2.1 Package installation is in NP

It is easy to see that PACKINST is in NP. Indeed, if R contains n packages, then a configuration α can be encoded in n bits and thus can be non-deterministically guessed in time proportional to n. Then, checking whether the constraints associated to each package are satisfied can be done in time linear in the size of the repository – for each package P in R such that $\alpha(P) = \text{Installed}$, we check that for all the disjunctive dependency clauses $P_1 \vee \cdots \vee P_k$ of P there exists an i such that $\alpha(P_i) = \text{Installed}$, and that for all packages P' conflicting with P we have $\alpha(P') = \text{Uninstalled}$. Since all those constraints appear explicitly in R and thus count in the size of R, the checking is done time in linear in the size of R (save the usual hidden logarithmic access factors, which may be superlinear in n).

3.2.2 Encoding a 3SAT instance as a Debian package installation

To see that PACKINST is NP-hard, we will show that any 3SAT problem can be reduced in polynomial time to an instance of of a Debian package installation problem.

Let $S = C_1 \wedge \ldots \wedge C_n$ be an instance of 3SAT, with each C_i being the disjunctions of three literals $(l_{i,1} \vee l_{i,2} \vee l_{i,3})$ each of the $l_{i,k}$ being either a propositional atom a or a negated propositional atom $\overline{a}$. Let $A = \{a_1, \ldots, a_k\}$ be the set of propositional atoms occurring in S. We build the following Debian repository R_S, containing a package P_S representing S itself, one package P_{C_i} for each clause C_i, and packages V_a, P_a, and $P_{\overline{a}}$ for each propositional atom a:

1. P_S depends $P_{C_1}, \ldots, P_{C_n}, V_{a_1}, \ldots, V_{a_k}$
2. P_{C_i} depends $P_{\ell_{i,1}} | P_{\ell_{i,2}} | P_{\ell_{i,3}}$, for each $1 \leq i \leq n$
3. V_a depends $P_a | P_{\overline{a}}$ for each atom a
4. P_a conflicts $P_{\overline{a}}$ for each atom a
5. $P_{\overline{a}}$ conflicts P_a for each atom a

The problem S is thus reduced to the instance $\langle R_S, P_S \rangle$ of PACKINST, which can be constructed in deterministic time polynomial in n. We will now show that $\langle R_S, P_S \rangle$ is a positive instance of PACKINST if and only if S is satisfiable. If we have a boolean valuation f satisfying S, this valuation gives us a configuration α whose constraints are all satisfied, and that maps P_S to $\mathrm{Installed}$, by taking $\alpha(P_S) = \alpha(P_{C_1}) = \cdots = \alpha(P_{C_n}) = \mathrm{Installed}$, $\alpha(P_a) = f(a)$ and $\alpha(P_{\overline{a}}) = \neg f(a)$ for every atom a. Therefore package P_S is indeed installable in the repository R_S.

Conversely, if $\langle R_S, P_S \rangle$ is a positive instance of PACKINST, there is a configuration α mapping P_S to $\mathrm{Installed}$ and satisfying all the constraints 1–4. Then we have that P_{C_i} and V_a must be mapped to $\mathrm{Installed}$ also (because of the dependency in 1) for every i and every atom a. By virtue of dependencies 3 and 4, for every propositional atom a exactly one of P_a and $P_{\overline{a}}$ must be installed. Furthermore, for every i, at least one of $P_{\ell_{i,k}}$ must be mapped to $\mathrm{Installed}$ because of the dependencies in 2. As a consequence, the valuation f, defined for every propositional atom a by $f(a) = \mathrm{true}$ if $\alpha(P_a) = \mathrm{Installed}$ and $f(a) = \mathrm{false}$ otherwise, satisfies S.

3.2.3 Encoding a 3SAT instance as an RPM package installation

The idea is the same as for the Debian encoding, but the details are slightly different because of the lack of explicit OR dependencies in the RPM format. We just give the RPM repository encoding the 3SAT instance S.

1. P_S depends $P_{C_1}, \ldots, P_{C_n}, V_{a_1}, \ldots, V_{a_k}$
2. P_a provides $C_{i_1}, \ldots, C_{i_k}$, for each $C_{i_1}, \ldots, C_{i_k}$ containing a literal equal to a
3. $P_{\overline{a}}$ provides $C_{i_1}, \ldots, C_{i_k}$, for each $C_{i_1}, \ldots, C_{i_k}$ containing a literal equal to $\overline{a}$
4. P_a provides V_a, for each atom a
5. $P_{\overline{a}}$ provides V_a, for each atom a
6. P_a conflicts $P_{\overline{a}}$, for each atom a

3.3 Conclusions

Despite the apparent differences, the constraint languages in DEB and RPM are sensibly equivalent in expressiveness, and the associated installation problems are both NP-complete.

This means that automatic package installation tools like APT, URPMI or SMART live dangerously on the edge of intractability, and must carefully apply heuristics that may be either safe (the approach advocated by SMART), and hence

still not guaranteed to avoid intractability, or unsafe, thus accepting the risk of not always finding a solution when it exists.

The detailed analysis of the algorithms underlying these existing tools, and a proposal for a state-of-the-art tool for automatic package management is one of the goals we set for the next deliverables.

Chapter 4

Proposals for improving the package management metadata

After extensively studying various packaging systems, and in particular the mainstream ones based on either the DEB format or the RPM format, one is led to the conclusion that these systems are used to manipulate and maintain the relationships among a wide range of informations that span conceptually very different levels of abstraction, as well as different periods in the lifetime of the resources contained in a package (i.e., compile-time, run-time, configuration and installation)

In this chapter we describe three different proposals for improving package management systems by refining and reorganizing the metadata used to describe software packages, with respect to the aspects we are concerned in this part of the EDOS project, i.e., dependency management.

The proposals embraces three different perspectives:

- Adding some features to the existing metadata in order to improve the expressivity of the language used to describe dependency relations.
- Adding metadata information to better specify, maintain and manipulate the relationships among conceptually different levels of abstraction which currently are treated and encoded in the same syntactic way.
- Separating and rationalizing these information in order to improve its management and to guarantee a backward compatibility with the existing tools.

The following sections detail these three proposals.

4.1 Increasing the expressive power of the dependency description language

We claim that the constraint language used in the package formats we examined is not expressive enough: we will substantiate this claim with a real-world example, and propose an elegant solution.

4.1.1 Expressivity shortcomings: building the OCaml packages

One real example of the shortcomings of the current dependency constraint language is provided by the whole family of binary packages built out of OCaml sources using the bytecode compiler.

The OCaml language is a type-safe language that can be compiled either in native code or in portable, architecture independent bytecode. All packages containing OCaml bytecode need the OCaml bytecode interpreter specific to the target architecture to be run, and it is then natural for them to contain a dependency towards a package like `ocaml-compiler-libs`. This is not very different from the the situation of many scripting languages, up to this point.

The difference appears when one discovers that, because of the design philosophy of the language, an OCaml program, like `Hevea`, compiled into bytecode using the compiler in some version n needs the runtime system of exactly the same version n to be executed.

Now, this kind of constraint is currently not expressible, and what the maintainer do is to manually hardwire the dependency on a particular version of the compiler into the source package, by writing something like the following statements in the control file (the names of the OCaml packages deviate from the ones used in the real Debian distribution for the sake of clarity):

```
Package: hevea

Build-Depends: ocaml-compiler == 3.08, ...

Depends: ocaml-runtime-system == 3.08, ...
```

Now suppose the version of the OCaml compiler is stepped up to 3.09; the maintainer is forced to update the control file for `hevea` (and all the other OCaml bytecode packages) as follows

```
Package: hevea

Build-Depends: ocaml-compiler == 3.09, ...

Depends: ocaml-runtime-system == 3.09, ...
```

This has two major disadvantages:

- we need to modify the source package, despite the fact that the source itself did not change at all
- the source package is now hardwired to a specific version of the compiler, so that somebody having another version of the OCaml compiler (because she still lives in the Debian stable distribution, for example), will be unable to recompile the package

4.1.2 A proposal: allowing binary constraints in dependencies

It seems clear that what the right approach would be to allow binary constraints (remember that they are not expressible via unary constraints!) in the language, so that one can write in the control file once and forall the connection that must exist between ocaml-compiler and ocaml-runtime-system: this means we can express binary relationships between package names and *variables* (like N in the following example) instead of just relationschips with constants as in the current systems.

```
Package: hevea
Build-Depends: ocaml-compiler == N, ...
Depends: ocaml-runtime-system == N, ...
```

One could also allow adding some extra unary constraint to the connection variables like N via a `Where:` tag as in

```
Package: hevea
Build-Depends: ocaml-compiler == N, ...
Depends: ocaml-runtime-system == N, ...
Where: N >= 3.0
```

which says that, whatever the version of OCaml we use, it must not only be the same for compiler and runtime, but must also be greater or equal to version 3.0.

This kind of constraints can be used in two ways:

- if the user just want to compile the package using the OCaml compiler already available on her system, the constraint will ensure that the right version of the runtimes is fetched and installed
- if the user wants to build a package for a given version of the runtime, the constraint will ensure that the right version of the compiler is fetched, installed and used

Note that these variables are real logical variables, in contrast to the "variables" currently existing in the control file of a Debian source package (see Section 2.1.3). The currently existing "variables" are mere placeholders for values to be filled in at package compilation time, and are not used to express constraints on package relations.

These variables could also be used to express more complex relations. For instance, a package like `Hevea` might depend on an additional package providing some library only for certain versions of the runtime system:

```
Package: hevea
...
Depends: ocaml-runtime-system == N, lib-ocaml-foo if N >= 3.5
```

As a side remark, notice that this extra power does not change the complexity class (the installation problem will still be NP-complete), but may be more costly in practice because more complex algorithms are needed to find a solution (we will need arc-consistency checks instead of node consistency).

4.2 Keeping semantically different metadata information separated

The current specification of both DEB and RPM package description format provides a very generic mechanism for describing package related metadata which enrich the standard inter-package relationships. This mechanism consists of the `provides` tag, which is present in both the DEB and RPM package description format, and which provides a totally generic and unstructured way of specifying additional package information that can be used when declaring package relationships.

As described in the previous section, the most natural way of using the `provides` tag is that of package group specification. By tagging a package using a `provides` clause, it is possible to logically associate that package to an identifier which will be then used by the package management system when trying to solve package dependencies. Of course, different packages can be associated to the same identifier, forming a group.

Even if this is a very powerful mechanism that adds a lot of flexibility to package metadata specification, it is very generic and unstructured and it is prone to tricky and undisciplined usages.

The trend in both RPM and DEB packages is to use (and abuse) the `provides` mechanisms in order to specify a wide range of useful metadata information regarding several unrelated semantical aspects.

In particular we have encountered that, above all in RPM packages, the `provides` tag is used to:

- Describe actual *abstract* (so called) capabilities provided by the package (e.g. `MTA` (Mail Transport Agent), `smptd`, `smtpdaemon`) that actually define package classes or groups.
- Describe some kinds of exported file-related information encoded as capability (e.g., `/bin/bash`)
- Describe some kind of capabilities not related to packages themselves but to the package management system (e.g., `rpmlib(VersionedDependencies)`)
- Describe some kind of structured information even if the capability information is flat (i.e., it is treated as a plain string not as a string representing a structured information). For example `perl(File::Find)`, `/bin/bash`, `libc.so.6(GLIBC_2.0)`

In particular what is written in the `provide` clause is often used to specify an arbitrarily encoded strings that further describe, in some way, the characteristics of a package.

For example, in the `apache-ssl-jserv` RPM package, there is the provided identifier `webserver` that tags the package as being part of the group of packages which exports the functionalities of a web server. The same package has even the `apache-ssl -jserv` identifier, that somehow, explicit that the package provides an *Apache* web server bundled with some additional functionalities (i.e. the `ssl` and `jserv` modules)

Often, the same type of information that could be reasonably included in metadata specification is encoded also in the package name. This is the case of many *server* or *modular* applications which can provide different kinds of functionalities, depending on how they are packaged or compiled:

- From a single source package we build several binary packages[1] by varying the build options (e.g., incorporating some modular options directly in the package itself, `kernel-2.4.9-3SGI_XFS_1.0.1.ia64.rpm`, `kernel-2.4.22-1.2199.nptl.i586.rpm`)

- Taking advantage of modular nature of the software we can combine and build a single package starting from different source distributions (e.g. `apache` and `jserv`), each one having its own independent evolution.

For example, we can find the following packages that are, basically, three flavors of the same web server application: `apache-ssl-jserv-1.3.2`, `apache-ssl-1.3.6`, `apache-1.3.7`.

The first one is a version of the *Apache* web server compiled with the `mod_jserv` and the `mod_ssl` extensions, the second one is a version of the same web server packaged only with the `mod_ssl` extension, and finally the third one is the plain version of the same web server again.

Encoding package information regarding something that can be still considered a capability in the package name is a widely used practice both in the RPM and in the DEB packages.

This fact poses the same dependency problems as for normal capabilities but at another level. This is true because what is encoded as an extension in the file name (e.g., `mod_ssl`) could be a *versioned* entity. In fact, many software are built in a modular and component oriented way, and the modules are often produced by someone else with their own versions, features etc. So we need to be able to specify metadata information even for modules and components bundled with a packaged software, in order to correctly reason on it.

A query on the package mentioned in the previous example executed on `http://rpm.pbone.net` reveals, in fact, that the `mod_ssl` bundled with the *Apache* web

[1] This is normally done, for example, by differentiating the `devel` version of a package containing a runtime library

server is at version `2.0.12-1.3.2`, and the `mod_jserv` is at version `0.9.11`. Unfortunately this information[2] is present in an unstructured format in the comment of the package itself and, from the point of view of an automated package management system, this information turns out to be completely useless

To partially solve this problem, one may decide that a module for an application should be packaged as a standalone package that can be installed on its own, but installing a module for an application often requires a modification of the configuration files of the same application, which is a dangerous and error-prone process.

While this problem is currently addressed and solved by making backups of the existing files, the approach taken by current package management systems does not take into account that there could be modules that, in some situations, simply could not be installed.

For example, if a server application is configured in a given way, an external module or an application which relies on a particular configuration of the server might not be installable. Since the information about the configuration of the application provided by a package is not exported explicitly by relevant metadata, what usually happens is that the package is nevertheless installed with the consequence of breaking the (good) configuration of the previously (working) applications.

At this point, it is clear that having a way for specifying in a *generic* way the metadata information gives a lot of flexibility for managing many of the previously described problems, but having a way for structuring such metadata information could be useful for increasing both the expressive power of the metadata provided with a given package and the reliability and effectiveness of actual package management systems.

This is the goal of the following sections where we describe with more detail the proposal to solve this kind of problems.

4.2.1 Additional metadata information

We have identified three main areas where it could be useful to have explicit metadata specification:

- *Definition of packages classes and grouping by capabilities*. This is the current standard usage of the `provides` tag for defining package classes, but the need to make this grouping more structured, using something similar to an ontology, is already apparent in the usage of the nonstandard field `Tag:` in some Debian packages like the recent versions of binutils

```
Package: binutils
Priority: standard
Section: devel
```

[2] http://rpm.pbone.net/index.php3/stat/4/idpl/38204/com/apache-ssl-jserv-1.3.2-2.i386.rpm.html

```
Installed-Size: 6004
Maintainer: James Troup <james@nocrew.org>
Architecture: i386
Version: 2.16.1-2
Provides: elf-binutils
Depends: libc6 (>= 2.3.2.ds1-21)
Suggests: binutils-doc (= 2.16.1-2)
Conflicts: gas, elf-binutils, modutils (<< 2.4.19-1)
Filename: pool/main/b/binutils/ \
          binutils_2.16.1-2_i386.deb
Size: 2377880
MD5sum: 37a46d934443c096e217aec8b2a2e303
Description: The GNU assembler, linker and binary
 utilities The programs in this package are used to
 assemble, link and manipulate binary and object files.
 They may be used in conjunction with a compiler and
 various libraries to build programs.
Build-Essential: yes
Tag: devel::machinecode, interface::commandline, \
     role::sw:shlib
```

- *Package compilation options*. This is a way of separating the information concerning what options have been built into the packaged software, giving them a clear semantic meaning. The information, that is currently present in an unstructured form both in the declared package `provides` and, often, even in the package file name, would be explicitly declared with a suitable expressive power, and made available for automatic processing of package management tasks. For example it would be possible to specify even the version of the built in modules (eg. `apache` version `1.3.2` with the module `jserv` version `0.9.11`).

- *Package configuration options*. By specifying such a metadata information it is possible to describe what are the configuration options that have been used to configure the packaged software. These configuration options might concern some of the *dynamic* attributes that characterize a given package. For example, with respect to the web server Apache, a configuration option will be the standard port where the server will listen to, or the default set of modules that will be activated when the server will start.

  Since what is written in a package metadata is a merely static information, this class of metadata description might seem useless. In fact, after installing a package, for example a web server, configured to be run on the port `80`, a system administrator could change that configuration parameter after the package installation. This would lead to an inconsistent information stored in the *installed* package database.

This problem could have different solutions:

- *Actually ignoring the problem* and treating, in case, the conflicts with respect to newly installed packages as *warnings* instead of *errors*. For example, if there is an installed web server configured to listen on the port `80`, if we try to install another package that needs port `80` we might simply issue a warning. It will be the responsibility of the system administrator to check that the web server is currently configured to run on the port `80` before actually installing the package.
- *Instrumenting* the packaged software with additional utilities which give the system administrator a way to change every configuration option *declared* in the package and, indirectly, updates the relative configuration options associated to the package stored in the installed package database. This will maintain the consistency between the current package configuration and what is recorded in the installed package database.

Notice that *configuration* and *compilation* options really need to be distinguished and may not be grouped in the same semantic class: while *compilation* options are hard-coded in the software and cannot be modified once the binary package has been built, *configuration* options describe some aspect of the software that can be changed at installation time, or even later. Their specification (see Section 4.2.3) have the same characteristics, but they are treated in different ways: *configuration* options may simply raise warnings when some dependency relations are not satisfied; *compilation* options, instead, produce fatal errors when they do not satisfy some of the dependency constraints.

4.2.2 Namespaces

In order to suitably exploit the additional metadata information that we outlined above, it is necessary to structure that information in an hierarchical way. Indeed, many configuration or compilation options could be present, with the same meaning, in different packages. This is the case, for example, of the `ssl` support that can be configured/enabled in several network-oriented software.

By introducing the concept of *namespace* it is possible to distinguish and associate a given option to a well defined context. For example it would be possible to declare `apache(ssl)` and `subversion(ssl)` and give a structured way to the other package to refer to these options.

The syntax for declaring a namespace is the following:

```
namespaceId(metadata)
```

This declaration will bind all the `metadata` to a particular namespace whose name is given by the identifier `namespaceId`

An implicit namespace is defined by the package name. So any metadata information regarding a particular package is implicitly defined in the namespace of that package.

4.2.3 Options specification

Configuration and compilation options have one of the follwing different types:

- *Boolean*: the option indicates a functionality that can be enabled or disabled.
- *Integer*: the option indicates a parameter that can be set to a specific integer value.
- *Generic*: the option indicates a declarative parameter that can be set to a generic (string) value.
- *Version*: the option indicates a versioned entity such as a module or a component that has been configured to be used with the given software.

The syntax for declaring those options is the following:

```
identifier[[:type]:=value]
```

where `type` is optional and can be one of the following: `bool`, `int` or `ver`. If the `type` information is missing, the option is treated as having a *generic* (string) type. If both `type` and `value` are missing then the declaration is a short version for a boolean option `id:bool:true`

Namespaces can be used in order to declare options for built-in or configured modules or components. Figure 4.1 shows some examples of option declarations.

`ssl` (or `ssl:bool:=true`)

(a) A boolean option

`port:int:=80`

(b) An integer option

`renderer:=opengl`

(c) A generic option

`modssl(version:ver:=0.9, md5, rsa, dsa)`

(d) An option bound to a namespace

Figure 4.1: Examples of option declarations

4.2.4 Option relations

Once options have been declared, it should be possible to establish relationships among them in the same way it is possible to establish relationships between packages and their versions.

Essentially the types of relations of relations that can be established among options are the same ones used with respect to package versions: `depends` and

`conflicts`. The first one denotes the option relations that must be satisfied, in order to correctly install the package; while the second one denotes the option relations that must not be satisfied in order to correctly install the package.

The operators we can use are the usual `=` and `!=`, which denote equality and inequality of option values (e.g., `ssl = false`, `port != 80`) and the other comparison operators <, <=, > and >=.

Obviously, since options are declared using a richer type system, the semantics of the previously described operators is dependent on the type of the options.

Moreover, since the difference between compilation and configuration options is just a matter of being immutable or not, when specifying option relations we do not differentiate between the two. It will be the package management system that when an option constraint isn't met will issue a warning in case of a configuration option or an error in case of a compilation option.

In order to specify option relations we use the following tags:

`Option-depends` and `Option-conflicts`

Namespace bound relations can be specified using the following syntax:

```
namespaceId(option rel value, ...)
```

In this case, all the relations will be verified against the options declared in that namespace. For example `Option-depends: apache(modssl(version=0.9, md5))` specify that the current package needs an option (module) `modssl` declared in the `apache` namespace. The `version` and `md5` options are bound to the `modssl` namespace. This example, finally, states that the current package needs an apache webserver configured with a modssl module whose version is `0.9` and it has been compiled with the support of the `md5` algorithm.

In the previous example `md5` is a shorthand for `md5:=true`. When a *boolean* option is mentioned in a relation but is not declared in neither `Configuration-options` nor in the `Compilation-options` it is assumed to be bound to the `false` value.

Finally, Figure 4.2 shows how options could be declared in the context of a package metadata specification.

In particular Figure 4.2(b) shows the options by using a namespace `modssl` for defining the way the optional module `modssl` has been configured, compiled and packaged with the Apache web server. Figure 4.2(c), instead, shows an independent `modssl` package that specify an `Options-conflict` with the `modssl` option defined in the `apache` namespace.

When specifying a `Options-depends` or `Options-conflict` relation we can refer to several options defined in a namespace. All the option relations specified must be satisfied by some configuration or compilation option declared in some package. If a package declares a superset of the options specified in the dependency relation, all the options that do not appear in the dependency relations are treated as *don't care*.

```
Package: apache
Version: 1.3.7
Configuration-options: port:int:=80
```
(a) Specification for a plain `Apache` web server

```
Package: apache-ssl
Version: 1.3.7
Configuration-options: port:int:=80
Compilation-options: modssl(version:version:=0.8, sha, md5)
```
(b) Specification for an *Apache* webserver compiled with modssl

```
Package: modssl
Version: 0.9
Compilation-options: sha, md5
Option-conflicts: apache(modssl)
```
(c) Specification for an independent `Apache modssl` module

Figure 4.2: Option specification

When we specify a generic option (a string) in a dependency relation it may refer either to a boolean option that must be set to true or to a given namespace that must have been declared. For example the `Options-depends: apache(modssl)` relation would be satisfied both by a package which declares a `modssl` option in the `apache` namespace, or by a package which declared no matter what options in a `apache(modssl(...))` namespace.

4.3 Separating metadata information

Most of the metadata information that is currently hard-coded in the package itself could be specified outside the package itself. This is particularly true above all with respect to the definition of package classes through the standard `provides` tag. Currently the class the package belongs to is defined by the package maintainer and is his own responsibility to choose the class identifier by actually hard-coding it in a static way in the package `provides` information. This is not a very flexible way to do this because:

- *Package classes might vary during time*: a package might be added or removed to a package class for various reasons .
- *Different systems might have different package class definitions*: what is a `web server` class for a distribution vendor might not be the same thing for another vendor and, moreover, the class identifier could be also named differently.

Hard-coding a package class as a capability directly into the package metadata information would determine, once and for all, the class the package (with that version) belongs to.

Separating this kind of information would, then, have those benefits:

- It will make possible to vary at later time the specification of the metadata information without altering the package it refer to (and, therefore, its version).
- Redistribute the responsibility of defining package related information among many persons instead of only the packager, i.e., give, for example, the vendor the power to decide what package belongs to what class for its own software distribution.
- As a consequence of the previous point, by separating and redistributing the roles in the package creation and maintenance process, the whole management is streamlined and made more efficient.

In order to do so, however, there should be an infrastructure where it is possible to store the metadata and the mappings towards actual packages. Such an infrastructure is shown in Figure 4.3. Package providers specify also metadata information and export it towards package management system by means of servers. Package providers may be either usual software distribution vendors (e.g., Mandriva, RedHat, etc.), or independent (and trusted) organizations which can provide themselves classifications for existing packages.

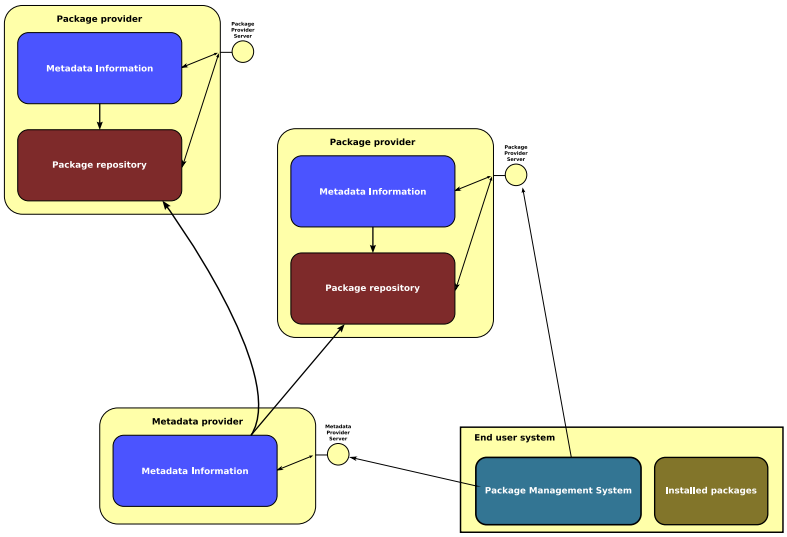

Figure 4.3: Metadata server infrastructure

4.3.1 Backward compatibility

This infrastructure does not require any modification to the current packages and their format. The information provided by the infrastructure can be used to complement or override the information that is hard-coded into package information metadata if the user requires so, but can be completely ignored otherwise.

4.3.2 Impact on the existing tools

The impact on existing tools is also minimal. During the package database information parsing phase, an existing tool could add to its internal representation the additional information coming from the external metadata servers specified by the users, after pre-processing the options into the internal representation of specially named `provides` tags. Once that is done, the usual algorithms for discovering a solution of the constraint could be used without modifications.

4.4 Remarks and related work

The previously described proposals entail several consequences. First of all, describing new metadata information poses an ontology design problem. On the other hand, building an infrastructure for supporting the new features of the package management and distribution system introduce the classic architectural problems that we have when we build complex and distributed systems.

Of course these problems are out of the scope of this deliverable and even of the relevant topics addressed by Work Package 2. However it is interesting to point them out.

In particular, with respect to the ontology perspective, we cite the AMOS project [**?**] which addresses the problem of *"building an ontology of open source code assets and a tool which helps the programmer to select, among all the described packages, those which are more promising for developing the desired software"*. This work is closely related to what we have described so far, even if it addresses a different (and more general) problem concerning software development and open source software categorization and search engines. However it could be interesting to investigate the adopted solutions in order to reuse them in our context.

On the distributed infrastructure side, instead, we must face the classic problems related to information distribution, synchronization and trust. Actually these topics are closely related to the ones addressed by the EDOS Project Work Package 4. This fact strengthen the relationships and the synergy between the different perspectives addressed by the EDOS Project.

Finally, it is worth to mention the W3C Member Submission regarding *Installable Unit Package Format Specification* [**?**]. This document describes an XML specification for describing installable packaged software units. Even if this initiative seems to completely overlap the problems we addressed, actually it simply

proposes a meta format in order *"describe a common installable unit package format [...] that is compatible with existing standard or de facto standard formats [...] and that encapsulates and uses the existing install technologies for the various hosting environments"*. In practice it doesn't address directly the problems we have with the description of fine grained features (such as package dependencies) but it only propose a generic and extensible way to describe high level package characteristics, leaving the responsibility of handling the actual problems to currently available technologies. Nevertheless, the design solutions proposed in this work, can be useful for refining the specification of the metadata information distributed using the infrastructure described in Section 4.3.

Chapter 5

Conclusions

We have provided an in-depth presentation of many largely used package management systems, focussing on the dependency management issues, which are central to this workpackage of the EDOS project. Based on this analysis, we have been able to pinpoint some limitations of the existent formats, and to make a concrete poroposal for a new metadata infrastructure that is backward compatible with the existing formats.

The total backward compatibility of our proposal, its low impact on the existing tools, and the separation of concerns between package managers and additional metadata maintainers we provide is an essential feature to give this proposal a chance of being accepted in the real world, as those that will adpot it will reap all the benefits without imposing any burden on the rest of the community.

We have also shown that the constraint languages used in DEB and RPM package description are sensibly equivalent, and that the associated installation problems are both NP-complete. Hence, automatic package installation tools like APT, URPMI or SMART live dangerously on the edge of intractability, and must carefully apply heuristics that may be either safe (the approach advocated by SMART), and hence not guaranteed to avoid intractability (in other words, in some cases the user may have to wait an exponential amount of time before getting an answer), or unsafe, thus guaranteeing an answer in limited time, but accepting the risk of not always finding a solution when it exists.

The detailed analysis of the algorithms underlying these existing tools, and a proposal for a state-of-the-art tool for automatic package management, both on the server side, to ensure consistency of whole package repositories, and on the client side, to ensure optimal management of a given installation, are now a clear necessity, and will be addressed in the next deliverables.